Technische Universität Ilmenau
Fakultät für Informatik und Automatisierung
Institut für Praktische Informatik und Medieninformatik
Fachgebiet Verteilte Systeme und Betriebssysteme

Diplomarbeit

Vergleich von monolithischen- und mikrokernbasierten Betriebssystemarchitekturen und deren Robustheits- und Echtzeiteigenschaften für den Einsatz in industrietauglichen Anwendungen

André Puschmann

November 2009

Technische Universität Ilmenau
Fakultät für Informatik und Automatisierung
Institut für Praktische Informatik und Medieninformatik
Fachgebiet Verteilte Systeme und Betriebssysteme

Diplomarbeit

Vergleich von monolithischen- und mikrokernbasierten Betriebssystemarchitekturen und deren Robustheits- und Echtzeiteigenschaften für den Einsatz in industrietauglichen Anwendungen

vorgelegt von:	André Puschmann
geboren am:	6. Dezember 1983 in Erfurt
Studiengang:	Informatik
Verantwortlicher Professor:	Prof. Dr.-Ing. habil. Winfried Kühnhauser
Betreuende wiss. Mitarbeiter:	Dr.-Ing. Hans-Albrecht Schindler
	Dipl.-Inf. Thomas Elste
Beginn der Arbeit:	11. Mai 2009
Abgabe der Arbeit:	11. November 2009
Registriernummer:	2009-11-11/107/IN03/2255

Erklärung

Hiermit erkläre ich, André Puschmann, dass ich die am heutigen Tag eingereichte Diplomarbeit selbstständig verfasst und ausschließlich die angegebenen Quellen und Hilfsmittel benutzt habe.

Ilmenau, 11. November 2009

Danksagung

Hiermit möchte ich mich bei allen bedanken, die zum Gelingen dieser Diplomarbeit beigetragen haben. Besonderer Dank gilt meinen Eltern, meinem Bruder sowie Janette, die lange auf die Fertigstellung dieser Arbeit warten musste.

Außerdem möchte ich mich bei allen Kollegen der Abteilung System-Design des IMMS Ilmenau, insbesondere bei meinem Betreuer Thomas, bedanken.

Für zahlreiche inspirierende Gespräche bedanke ich mich ebenfalls bei den Mitarbeitern des Fachgebiets Betriebssysteme der TU-Ilmenau sowie bei Adam Lackorzynski, der mir an der einen oder anderen Stelle sehr behilflich war.

All dies wäre nur schwer ohne meine Freunde möglich gewesen. Daher möchte ich mich an dieser Stelle bei allen bedanken:
`MD5 (Freunde) 96b04f26ddac308d4f26957d8b59fd9f`

Inhaltsverzeichnis

Kapitel 1

1 Einleitung

Sie sind in unserem täglichen Leben allgegenwärtig, jeder nutzt sie, ist oft sogar von ihnen abhängig, ob bewusst oder unbewusst. Die Rede ist von Computersystemen. Sie sind in allen Bereichen zu finden: in Mobiltelefonen, in Bankautomaten oder im Automobil. Besonders aber in industriellen Prozessen - bei der Produktion von Handelsgütern oder bei der Erbringung von Dienstleistungen - wird ihre Ausdauer und Präzision hoch geschätzt. Ein besonderes Alleinstellungsmerkmal sind deren lange Produktlebenszyklen. Während die Halbwertzeit von IT-Systemen im Endanwendermarkt oft nur Monate beträgt, so verrichten Computersysteme im industriellen Umfeld häufig für Jahre oder gar Jahrzehnte ihren Dienst. Sie erfüllen zudem nicht selten kritische Aufgaben, sodass Ausfälle hohen monetären oder (ungleich schlimmer) menschlichen Schaden verursachen können. Verlässlichkeit ist in dieser Anwendungsdomäne daher die Kernanforderung.

Im allgemeinen Sprachgebrauch wird der Begriff Verlässlichkeit meist jedoch ausschließlich mit der verwendeten Hardware assoziiert. Neben der Resistenz gegenüber Umwelteinflüssen wie erhöhte Temperatur, Staub oder Feuchtigkeit, ist es für industriell genutzte Systeme ebenso wichtig, auch vor Fehlern in der Betriebssoftware geschützt zu sein. Diese umfasst, neben Anwendungssoftware, welche die Geschäftslogik implementiert, selbstverständlich auch das darunter liegende Betriebssystem.

Auf dem steinigen Weg, hin zu ausfallsicheren Computern, haben Entwickler grundsätzlich zwei Möglichkeiten: Fehler gar nicht erst zu begehen oder sie nach deren Auftreten zu beheben. Ein vollkommen fehlerfreies System ist nur schwer vorstellbar.

Ziel der Bemühungen ist daher, die Ausbreitung von Fehlern und den durch sie verursachten Schaden zu begrenzen. Diesen Anforderungen kann in weitverbreiteten, monolithisch konzipierten Betriebssystemarchitekturen nur schwer entsprochen werden. Durch enge Bindung der Systemkomponenten untereinander kann ein einziger Fehler hier leicht die Stabilität des Gesamtsystems gefährden. Der Umstand, dass praktisch das vollständige Betriebssystem mit uneingeschränkten Privilegien ausgeführt wird, intensiviert diese Gefahr weiter.

Eine Möglichkeit, diesen Problemen zu begegnen, könnte im Einsatz von mikrokernbasierten Betriebssystemarchitekturen bestehen. Nur die notwendigsten Primitive werden hier von einem schlanken Systemkern zur Verfügung gestellt. Der Großteil aller Systemservices wird, auf dem Mikrokern aufbauend, im Userspace mit herabgesetzten Privilegien implementiert. Projekte wie μSina [1] konnten die Tauglichkeit von Systemen dieser Art bereits im Bereich sicherheitskritischer Anwendungen demonstrieren. Doch sind sie auch in der Lage, enge zeitliche Vorgaben in industriellen Anwendungen zu erfüllen? Was unterscheidet sie in dieser Hinsicht von monolithischen Systemen?

Schwerpunktthema dieser Arbeit ist es, die grundsätzliche Tauglichkeit von mikrokernbasierten Betriebssystemen für den Einsatz in industriellen Anwendungen zu analysieren. Den Mittelpunkt der Betrachtungen bildet dabei die Untersuchung von Verlässlichkeits- und Echtzeitaspekten, im Vergleich zu monolithisch konzipierten Lösungen. Die Analyse, als auch die anschließende Evaluation der Kernparameter, soll dabei auf Basis zwei verbreiteter Vertreter beider Architekturkonzepte stattfinden. Zur Überprüfung der zu Beginn formulierten Annahme sowie zur Unterstützung der Evaluierung wurde beispielhaft ein ursprünglich für Linux entwickelter EtherCAT-Stack auf ein mikrokernbasiertes System portiert. Die Beschreibung des entstandenen Systems als auch die dabei überwundenen Hürden sind ebenfalls Gegenstand dieser Arbeit.

Aufbau der Arbeit

Zum besseren Verständnis und um den Leser einen Überblick zur vorliegenden Arbeit zu ermöglichen, soll in diesem Abschnitt deren Aufbau kurz erläutert werden.

In Kapitel 2 wird der Leser zunächst in essenzielle Grundlagen eingeführt. Im Anschluss daran werden in Kapitel 3 dem Thema verwandte Arbeiten diskutiert.

Aufbauend auf dem bis dahin erörterten theoretischen Basiswissen, strukturiert sich die weitere Arbeit in zwei Hauptthemengebiete. Der erste Teil, welcher eher konzeptionell-analytisch orientiert ist, widmet sich der Untersuchung von Echtzeit- und Verlässlichkeitseigenschaften in monolithischen sowie mikrokernbasierten Betriebssystemarchitekturen. Beide Aspekte werden hierfür zunächst ausführlich einzeln betrachtet. Im Anschluss daran folgt eine wechselseitige Diskussion. Zur besseren Orientierung basiert der Rest der Arbeit ebenfalls auf dieser Grundstruktur.

Im zweiten Teil werden die zuvor theoretisch betrachteten Aspekte experimentell evaluiert. Hierzu wird zunächst in Kapitel 5 der Entwurf sowie die prototypische Implementierung eines mikrokernbasierten EtherCAT-Masters dargestellt. Kapitel 6 bewertet diese auf Grundlage zuvor festgelegter Metriken.

Kapitel 7 fasst schließlich die gewonnenen Erkenntnisse zusammen und liefert einen Ausblick auf zukünftige Entwicklungen.

Kapitel 2

2 Grundlagen

Dieses Kapitel führt den Leser in die theoretischen Grundlagen der Arbeit ein. Zu Beginn werden die wesentlichen Begrifflichkeiten in Echtzeitsystemen behandelt (Abschnitt 2.1). Anschließend wird näher auf die Anforderungen an Echtzeitsysteme eingegangen, um danach Faktoren zu ermitteln, die das zeitliche Verhalten beeinflussen können.

Im Abschnitt 2.2 erfolgt eine nähere Betrachtung verlässlicher Computersysteme. Dabei werden anfänglich einige Grundbegriffe erörtert, da diese im allgemeinen Sprachgebrauch, aber auch in der Fachliteratur, oft nicht präzise unterschieden werden. Des Weiteren wird der Leser in diesem Teil u.a. in die Themen *Fehlererkennung*, *Fehlerklassifizierung* sowie in die *Fehlerevaluation* eingeführt. Abschnitt 2.3 erläutert die grundlegende Idee von Mikrokernen. Außerdem werden der L4-basierte Mikrokern *Fiasco* und das darauf aufbauende Betriebssystem *DROPS* vorgestellt, da dies das zugrunde liegende System für die anschließende experimentelle Analyse ist. Der letzte Teil dieses Kapitels (Abschnitt 2.4) widmet sich dem Feldbussystem *EtherCAT*.

2.1 Echtzeitsysteme

Ein Echtzeit-Computer-System ist ein Computer-System, bei dem korrektes Systemverhalten nicht ausschließlich über die logisch korrekte Berechnung von Ergebnissen bestimmt wird. Vielmehr kommt es ebenso darauf an, wann diese Ergebnisse produziert werden. Die Kombination von funktionalen und zeitlichen Anforderungen

ist eine wesentliche Eigenschaft von Echtzeitsystemen. Da solche Systeme zumeist auch sicherheitskritische Aufgaben übernehmen (im Sinne von *Safety*), werden an diese auch oft Anforderungen bzgl. deren Verlässlichkeit gestellt. Die gleichzeitige Erfüllung aller drei Hauptanforderungen stellt dabei eine besondere Herausforderung dar (siehe Abbildung 2.1).

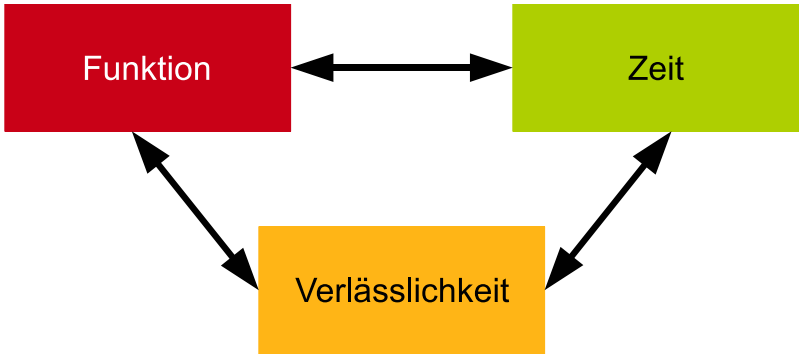


Abb. 2.1: Spannungsdreieck Echtzeitanforderungen

Klassifikation

Echtzeitsysteme müssen innerhalb einer bestimmten, von der Applikation oder Umgebung vorgegebenen Zeit, auf externe oder interne Stimuli reagieren können. Der Zeitpunkt, an dem die Ausgabe vorliegen muss, wird Frist oder *Deadline* genannt. Ist die erzeugte Ausgabe auch nach Überschreiten der Frist noch von Nutzen, so wird diese auch als weich oder *soft* bezeichnet. Hat das Ergebnis nach Ablaufen der Frist keine Bedeutung mehr, dann nennt man diese hart (*hard*) oder *firm*. Die letztgenannten Typen unterscheiden sich nun bzgl. der Auswirkungen, die ein eventuelles Verfehlen der Frist verursachen kann. Hat das Verpassen möglicherweise katastrophale Auswirkungen (bspw. einen Verkehrsunfall oder einen Flugzeugabsturz), dann spricht man von einer harten Frist.

Systeme, die mindestens eine harte Frist zu verantworten haben, werden harte Echtzeitsysteme genannt. Falls keine harten Anforderungen bzgl. des zeitlichen Verhaltens existieren, so nennt man die Systeme auch weiche Echtzeitsysteme (vgl. [2], S. 3). Die Unterscheidung verschiedener Systeme anhand zeitlicher Anforderungen bildet die Grundlage für alle weiteren Entwurfsentscheidungen bei deren Entwicklung.

Latenz und Jitter

Zwei weitere Grundbegriffe, die u.a. bei Echtzeitsystemen eine große Rolle spielen, sind *Latenz* und *Jitter*. Ersterer beschreibt dabei grundsätzlich die Verzögerung von dem Moment, an dem ein bestimmtes Ereignis auftritt (z. B. Auslösen eines Interrupts), bis zu dem Moment, an dem das Ereignis erkannt wird bzw. darauf reagiert werden kann (z. B. erste Instruktion im Interrupt-Handler). Die in diesem Beispiel beschriebene Verzögerung wird auch Interrupt-Latenz eines Betriebssystems genannt.

Ein weiteres Beispiel ist die Scheduling-Latenz. Diese beschreibt den Zeitbereich, beginnend vom Moment, an dem ein Task in den Zustand *Bereit* wechselt, bis zu dem Moment, an dem er tatsächlich ausgeführt wird, sich also im Zustand *Rechnend* befindet.

Der Begriff Jitter definiert die Variabilität oder Schwankung einer gewissen Größe, auf die sich bezogen wird. Er ist somit ein Maß für die Genauigkeit einer Größe (z. B. der Interrupt-Latenz). Ein kleiner, maximaler Jitter drückt auch eine kleine, maximale Abweichung aus.

Obwohl der Begriff Latenz beim Design von Echtzeitsystemen eine wichtige Stellung einnimmt, lassen sich hierdurch nur Aussagen hinsichtlich der Performanz oder des möglichen Durchsatzes eines Systems treffen. Ausgehend davon ist der Jitter für die Bewertung der Echtzeitfähigkeit ausschlaggebend, da dieser Zeitgrößen oder Zeitbereiche nach oben oder unten begrenzt.

Anforderungen

Primäres Ziel beim Entwurf von Echtzeitsystemen ist Vorhersagbarkeit bzw. Determinismus. Zeitliche Anforderungen sollen garantiert und pünktlich erfüllt werden. Beim Systemdesign sollten daher alle Schichten der Architektur betrachtet werden. Dies beginnt bei der eingesetzten Hardware, betrifft den Betriebssystemkern, reicht über die darauf aufbauenden Systemservices und endet schließlich beim Anwendungsprogramm.

Echtzeitsysteme werden fälschlicherweise oft damit verknüpft, eine gewisse Tätigkeit besonders schnell ausführen zu müssen. Vielmehr geht es jedoch darum, Aufgaben

rechtzeitig auszuführen. Der Begriff „Rechtzeitsystem" erscheint in Anlehnung an dieses Ziel treffender.

Wie eingangs bereits erwähnt, sind Echtzeitsysteme oft auch sicherheitskritische Systeme. Diese Anwendungsdomäne erfordert typischerweise weitere nicht-funktionale Eigenschaften, wie z. B. eine kontrollierte Bearbeitung von fehlerhaften oder nicht erwarteten Eingaben, um möglicherweise katastrophale Folgen zu verhindern.

Evaluation

Die grundlegenden Begriffe Frist, Latenz und Jitter bilden die Basis für die Bestimmung der Ausführungszeit (z. B. einer Code-Routine) im schlimmsten zu erwartenden Fall (*worst case execution time* oder *WCET*). Eine enorme Herausforderung beim Design von Echtzeitsystemen stellt die Abschätzung der WCET dar. Diese darf dabei nicht unterschätzt werden, damit die geforderte Frist in jedem Fall nicht überschritten wird. Sie darf aber auch nicht zu großzügig bemessen werden, damit nicht sinnlos Ressourcen verschenkt oder gar Applikation als nicht realisierbar eingestuft werden, die bei genauerer WCET-Abschätzung doch umsetzbar wären.

Im Gegensatz zur WCET, welche Aussagen über Anwendungsprogramme trifft, beschreibt der *WCAO* (*worst case administrative overhead*) Zeitverzögerungen, die durch das Betriebssystem oder Betriebssystemservices verursacht werden. Dies schließt Einflüsse ein, die ein Anwendungsprogramm zwar direkt betreffen, aber durch dieses nicht beeinflusst werden können, wie zum Beispiel Kontextwechsel, Scheduling, Cache/TLB laden, Interrupts oder Direct-Memory-Access (vgl. [2], S. 86). Darüber hinaus können weitere Anwendungsprogramme oder andere Systemaktivitäten zusätzliche Last verursachen.

Einflussfaktoren

Bei der Realisierung von Echtzeitsystemen spielen eine Reihe von Störgrößen eine wichtige Rolle. Diese können entscheidenden Einfluss auf das Antwortverhalten des Systems nehmen und somit die Vorhersagbarkeit bzw. die Erfüllung der Garantien gefährden. Ein generelles Problem beim Design von Echtzeitanwendungen stellt die (große) Lücke zwischen Normalzustand (*average case*) und dem schlimmsten

anzunehmenden Zustand (*worst case*) dar. Darüber hinaus verschlingen zu großzügig bemessene Reserven, die zur Garantie des schlechtesten Falls eingeplant werden, oft wertvolle Ressourcen.

Zur weiteren Untersuchung soll nun eine Unterteilung der Einflussfaktoren vorgenommen werden. Hierzu wird das zu untersuchende Echtzeitsystem zunächst in eine Hard- und Softwarekomponente aufgespalten. Die weitere Analyse der Architekturmodelle wird durch eine Gliederung der Software in betriebssystem- bzw. anwendungsspezifische Faktoren unterstützt. Abbildung 2.2 verdeutlicht dieses Schema und stellt die im Folgenden näher erläuterten Faktoren grafisch dar.

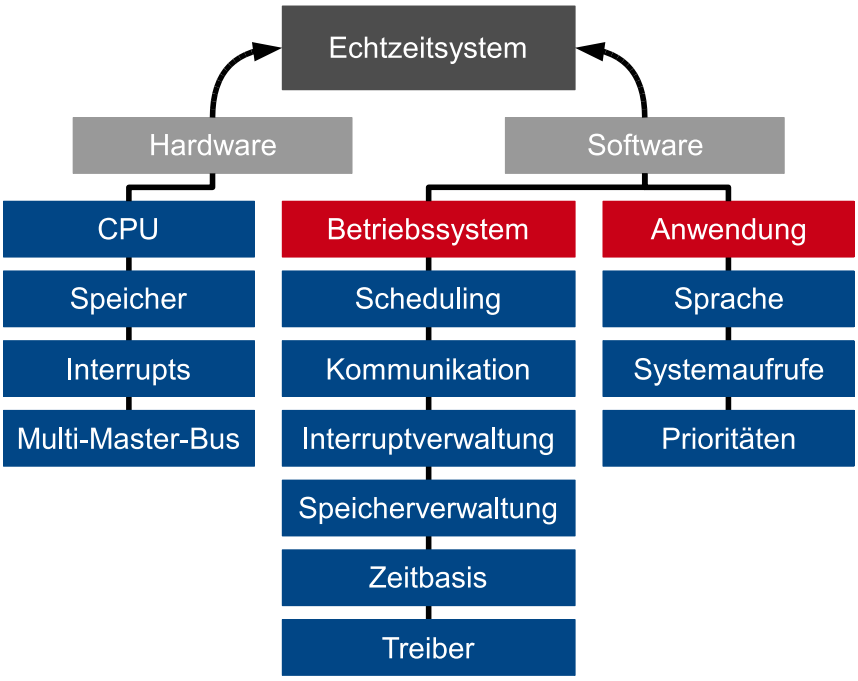


Abb. 2.2: Faktoren, die das zeitliche Verhalten von Computersystemen beeinflussen

Hardware

Der linke Ast des Baumes verdeutlicht, dass der Einsatz moderner Hardware die zeitliche Unvorhersehbarkeit des Gesamtsystems verstärkt (vgl. [3]). CPU-Caches, mehrstufige Prozessorpipelines oder *Out-of-Order-Execution* sind Techniken, die zur Steigerung der Performanz entwickelt wurden. Dieses Ziel wird im Mittel auch

erreicht, kann bei der Anwendung in Echtzeitsystemen jedoch zu teilweise starken Schwankungen bei der Ausführungsgeschwindigkeit führen und verschlechtert somit die Vorhersagbarkeit. In neueren x86-basierten Rechnern kommt oft zusätzlich eine Technologie zum Selbstschutz der Hardware zum Einsatz (*System-Management-Mode* oder SMM). Dieser soll zum Beispiel vor Beschädigung durch Überhitzung schützen, kann vom Betriebssystem jedoch nicht kontrolliert werden und ist somit potenziell für unakzeptable Verzögerungen verantwortlich.

Auch Speicherzugriffe können das Verhalten von Echtzeitsystemen empfindlich stören. Besonders leere Caches und *Translation Lookaside Buffer* (TLB) führen hierbei zu erheblichen Verzögerungen.

Betriebssystem Neben einigen, durch das Betriebssystem nicht beeinflussbaren Komponenten, existieren auch eine Reihe weiterer Faktoren, auf die per Software Einfluss genommen werden kann. Die wichtigsten gemeinsam genutzten Ressourcen in Computersystemen sind Rechenzeit und Speicher. Erstere wird vom Scheduler verteilt. In Echtzeitsystemen kommen dabei Algorithmen zum Einsatz, die sich deutlich von denen in gewöhnlichen Computersystemen unterscheiden.

Beim Scheduling von voneinander abhängigen Threads muss zusätzlich darauf geachtet werden, dass beispielsweise durch den blockierenden Zugriff auf eine gemeinsam genutzte Ressource nicht das Problem der (unkontrollierten) Prioritätsinversion (*priority inversion*) entsteht (vgl. [2], S. 234).

In Betriebssystemen mit dynamischer Speicherverwaltung können Programme zur Laufzeit Speicher anfordern und diesen auch wieder freigeben. Da die Allokation von Speicher im Betrieb fehlschlagen kann, wird dies in vielen sicherheitskritischen Anwendungsdomänen (bspw. in der Automobilbranche) strikt verboten. In solchen Systemen kann also nur auf statischen, zur Übersetzungszeit festgelegten, Speicher zugegriffen werden. Um zumindest die zeitlichen Anforderungen in Echtzeitsystemen gewährleisten zu können, sollte, falls dynamischer Speicher erwünscht oder gar erforderlich ist, auf einen echtzeitfähigen Speicherverwaltungsalgorithmus zurückgegriffen werden. TLSF [4] ist ein in letzter Zeit häufig referenzierter Algorithmus, der diese Eigenschaften bietet. Des Weiteren sollte die dynamische Auslagerung von Speicherseiten (*paging*) in Echtzeittasks deaktiviert werden (*memory pinning/locking*).

Der CPU-Cache ist ein Zwischenspeicher des Prozessors, der Zugriffe auf den wesentlich langsameren Hauptspeicher puffert. Aufgrund seiner begrenzten Größe finden nicht alle vom Prozessor benötigten Befehle oder Daten in ihm Platz. Fordert die CPU nun Daten vom Cache an, die gerade nicht gespeichert sind, so muss dieser die benötigten Inhalte erst aus dem Hauptspeicher laden, was natürlich wesentlich mehr Zeit in Anspruch nimmt. Verfahren wie Cache-Partitionierung [5] verfolgen das Ziel, die gegenseitige Beeinflussung von mehreren konkurrierenden Applikationen bzgl. der Benutzung des Caches zu mindern. Die Idee ist, den Arbeitsspeicher in verschiedene Regionen (Farben) aufzuteilen. Die Regionen sind dann jeweils bestimmten Cache-Bereichen (*cache lines*) zugeordnet. Erfolgt nun eine geschickte Zuweisung des Speichers an die Applikationen, so kann, transparent für die Applikation selbst, eine Beeinflussung durch andere Anwendungen ausgeschlossen werden.

Der Austausch von Nachrichten ist in Echtzeitsystemen fast ausnahmslos notwendig, ob zwischen Anwendungen innerhalb eines Computersystems (*inter process communication*, *IPC*) oder mit externen Subsystemen. Das Betriebssystem, verwendete Treiber, aber auch die Hardware, die möglicherweise zum Aufbau von Kommunikationsnetzen von Nöten ist, sollten daher deterministisches Verhalten aufweisen.

Um Timer und andere zeitabhängige Dienste bereitzustellen, benötigt jedes Betriebssystem eine akkurate Zeitbasis. Diese wird meist mit Hardwareunterstützung, z. B. von einer Echtzeituhr (RTC), periodisch oder im Einzelbetrieb (*one shot mode*) generiert. Der Takt der internen Uhr bestimmt somit auch die Granularität, mit der ein System auf Timerereignisse reagieren kann.

Anwendung Auch auf Anwendungsebene kann Einfluss auf das Echtzeitverhalten eines Computersystems genommen werden. Oft werden hier nur Politiken vorgegeben (z. B. Threadpriorität setzen), welche dann wiederum vom Betriebssystem umgesetzt werden (z. B. Scheduling durchführen). Für den Anwendungsentwickler ist es notwendig, sich sehr genau mit den bereitgestellten Schnittstellen auszukennen und diese auch gezielt zu verwenden. Dies gilt gerade in Systemen, in denen zeitkritische und weniger zeitkritische Anwendungen parallel operieren und daher nicht alle Systemaufrufe konsequent auf Echtzeitverhalten optimiert sind (z. B. nicht jeder Systemaufruf gestattet es, eine maximale Wartezeit zu übergeben).

Auch die Besonderheiten oder Sprachkonstrukte einer Programmiersprache können

Einfluss auf das zeitliche Verhalten nehmen (z. B. *Garbagecollector*). Echtzeitsysteme werden oft in C/C++ oder ADA entwickelt. Es existieren aber auch Echtzeiterweiterungen für die Hochsprache Java. Bei der Verwendung dieser wird die Nutzung von Speicher allerdings enorm eingeschränkt.

Echtzeitbetriebssysteme

Unter einem Echtzeitbetriebssystem (*real time operating system*, *RTOS*) versteht man ganz allgemein ein Betriebssystem, welches in der Lage ist, zeitliche Garantien für den bereitgestellten Service einzuhalten. Dabei existiert eine Vielzahl von unterschiedlichen Ausprägungen, je nachdem für welchen Zweck oder in welcher Umgebung das System eingesetzt wird. Die Spannweite reicht dabei von einfachen Threadbibliotheken auf Einprozessorsystemen bis hin zu mächtigen Betriebssystemen mit getrennten Adressräumen auf Mehrprozessorrechnern. Erstere kommen hauptsächlich in 8- oder 16-Bit Mikrokontrollern ohne *MMU* (*memory management unit*) zum Einsatz. Diese kennen nur einen gültigen Adressraum, in dem alle Threads ablaufen. FreeRTOS [6] ist bspw. ein Vertreter dieser Art von Betriebssystemen.

Die angesprochene Unterteilung gilt nicht ausschließlich für Echtzeitbetriebssysteme. Heute ist es allerdings so, dass Mikrokontroller der unteren Leistungsklasse annähernd nur noch in eingebetteten Systemen Verwendung finden. Hier spielen zeitliche Anforderungen oft eine entscheidende Rolle, sodass in diesem Zusammenhang nahezu ausnahmslos von Echtzeitsystemen gesprochen wird.

In leistungsfähigeren Systemen ab 32-Bit Wortbreite ist häufig eine MMU zu finden. Diese Rechner können daher Anwendungen in unterschiedlichen Adressräumen ablaufen lassen. Der Betriebssystemkern ist hier deutlich komplexer. Auch die Kommunikation ist aufwendiger, da beim Nachrichtenaustausch unterschiedlicher Tasks der Adressraum umgeschaltet werden muss. Bekannte Vertreter dieser Betriebssystemklasse sind bspw. Windows, Linux oder auch L4-basierte Systeme.

2.2 Verlässliche Systeme

Verlässliche Computersysteme (*dependable computer systems*, *dependability*) sind Computersysteme, die in der Lage sind, einen Service zu liefern, welchem der Benutzer

des Systems mit Recht vertrauen kann ("that can justifiably be trusted", [7], S. 2). Die erwartete Funktionalität wird dabei von der Funktionsspezifikation beschrieben. Wird sie erfüllt, so arbeitet das System korrekt. Weicht der erbrachte Service jedoch von der spezifizierten Funktionalität ab, so liegt ein Ausfall (*failure*) vor. Als Irrtum (*error*) bezeichnet man den Zustand des Systems, der zu einem Ausfall führen kann (aber nicht muss). Ein Ausfall liegt erst vor, wenn ein Irrtum die Schnittstelle einer Komponente verlässt und somit die Funktionalität des Systems beeinträchtigt. Ein *fault* (auch Defekt oder nur Fehler) ist die Ursache für einen Irrtum.

Leider kann das Wort *Fehler* in der deutschen Sprache als Synonym für jeden dieser drei Begriffe genutzt werden. Um die jeweils korrekte Bedeutung hervorzuheben, wird diesem daher in der vorliegenden Arbeit die englische Übersetzung nachgestellt. Abbildung 2.3 verdeutlicht deren Zusammenhang.

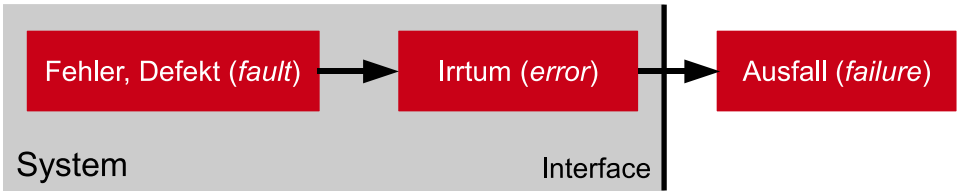


Abb. 2.3: Zusammenhang zwischen Fehler, Irrtum und Ausfall (Fehlerkette)

Die drei genannten Faktoren bezeichnet man auch als die Bedrohungen (*threats*), denen verlässliche Computersysteme ausgesetzt sind. Im Grundlagenwerk [7] beschreibt Laprie et al. weiter die Eigenschaften (*attributes*) von verlässlichen Systemen, sowie die Hilfsmittel (*means*) mit denen diese entworfen werden können. Laprie unterteilt Letztere in Fehlerprävention (*fault prevention*), Fehlertoleranz (*fault tolerance*), Fehlerentfernung (*fault removal*) sowie Fehlervorhersage (*fault forecasting*).

Wie in Abbildung 2.4 dargestellt, werden die vier Maßnahmen zur weiteren Analyse in Fehlervermeidung (*fault avoidance*) bzw. Fehlerbehandlung (*fault handling*) klassifiziert. Methoden zur Fehlervermeidung beschreiben dabei passive Verfahren, die z. T. auch offline, also vor Inbetriebnahme des Systems, durchgeführt werden können. Diese umfassen folglich Mechanismen der Fehlerprevention, der Fehlervorhersage sowie Methoden zur Fehlerentfernung während der Entwicklungsphase. Fehlerbehandlung beinhaltet stattdessen Maßnahmen zur aktiven Verbesserung der Systemverlässlich-

keit. Diese werden z. T. online, also während der Operationsphase, durchgeführt. Die Fehlerbehandlung fasst Techniken zur Fehlertoleranz sowie zur Fehlerentfernung (während der Ausführung) zusammen.

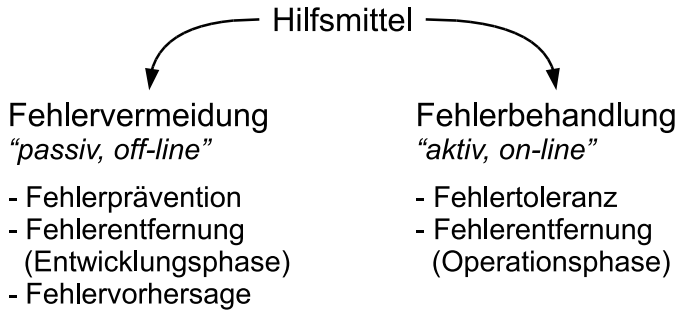


Abb. 2.4: Hilfsmittel zur Verbesserung der Verlässlichkeit

Robustheit und Fehlertoleranz

In der Literatur wird leider oft nicht präzise zwischen Fehlertoleranz und Robustheit unterschieden. Umgangssprachlich bezeichnet man ein System als robust, wenn es in der Lage ist, die ihm gestellten Anforderungen verlässlich zu erfüllen. Beispielsweise soll der Rohbau eines Hauses in der Lage sein, alle weiteren Bauelemente (z. B. das Dach) aufzunehmen. Er bildet somit die Basis für ein stabiles Bauwerk (Service oder Funktionalität). In der Realität ist es jedoch leider so, dass eine ganze Reihe von Störgrößen auf das System einwirken. Es ist bspw. vorstellbar, dass zu starker Wind einen tragenden Pfeiler beschädigt (Defekt), somit die Stabilität das Haus gefährdet (Irrtum) und es zum Einsturz bringen könnte (Ausfall).

Um die an das Haus gestellten Anforderungen bzgl. Verlässlichkeit erfüllen zu können, haben Bauingenieure und Architekten grundsätzlich, wie bereits oben erwähnt, zwei Möglichkeiten. Man könnte beispielsweise versuchen, Fehler gänzlich zu vermeiden. Würde man eine Glaskuppel um das Haus herum bauen und somit sicherstellen, dass der Wind den Pfeiler nicht beschädigen kann, so könnte das Haus auch nicht zum Einsturz gebracht werden. Dieses einfache Beispiel soll verdeutlichen, dass die gänzliche Vermeidung von Fehlern in nicht-trivialen Systemen oft nicht durchführbar ist.

Fehler können nicht restlos ausgeschlossen werden. Das System muss also in der Lage sein, mit (bekannten oder erwarteten) Fehlern umzugehen und auch nach deren Auftreten, seinen Service, bis zu einem gewissen Level, aufrecht zu erhalten. Eine solche Eigenschaft wird als Fehlertoleranz (*fault tolerance*) bezeichnet. Im Gegensatz dazu beschreibt der Begriff Robustheit (*robustness*) die Fähigkeit, selbst nach Auftreten unerwarteter oder unvorhersehbarer Störungen, einen Service (bis zu einem gewissen Grad) bereitzustellen (vgl. [8], S. 2).

Neu zu entwickelnde Systeme werden, möglichst vor dem Beginn der Implementierung, genau spezifiziert. Dabei sollten der Normalzustand des Systems, alle korrekten Ein- und Ausgaben sowie der Fehlerzustand bzw. die fehlerhaften Ein- und Ausgaben betrachtet werden. Letzteren Teil bezeichnet man auch als Fehlerhypothese, also die Spezifikation von Typ und Anzahl der Fehler, die das System zu tolerieren hat. Ein fehlertolerantes System kann demnach immer nur die Fehler behandeln, die während der Anforderungsspezifikation in der Fehlerhypothese festgehalten wurden (vgl. [2], S. 73).

Um ein Haus beispielsweise vor höheren Windgeschwindigkeiten (als normal erwartet) zu schützen, wird vom Bauingenieur bei der Berechnung der Statik eine Annahme darüber getroffen, wie hoch die Windgeschwindigkeit am Standort des Hauses maximal steigt. Daraufhin wird das Haus mit Stahlträgern verstärkt, um dasselbe tolerant gegenüber den maximal zu erwartenden Windgeschwindigkeiten zu machen. Lassen sich Windgeschwindigkeit oder andere mögliche Störgrößen nicht genau berechnen oder bestimmen, so können zusätzliche Stahlträger die Robustheit des Hauses erhöhen. Dieses bildliche Beispiel verdeutlicht weiter, dass eine eindeutige Zuordnung, ob ein bestimmter Mechanismus nun eine Fehlertoleranzmaßnahme ist oder die Robustheit eines Systems verstärkt, oft nicht vollzogen werden kann.

Evaluation

Bevor verlässliche Computersysteme eingesetzt werden, müssen sie zuvor eine Reihe von Funktionsprüfungen absolvieren. Neben den funktionalen und zeitlichen Anforderungen wird dabei auch die Verlässlichkeit des Systems evaluiert. Da bestimmte Fehler sehr selten auftreten, würden Funktionsprüfungen im Feld erhebliche Zeit in Anspruch nehmen. Daher wird oft eine Technik namens *Fault-Injection* verwendet.

Bei dieser werden kontrolliert Defekte in ganze Systeme oder einzelne Komponenten eingeschleust und deren Verhalten daraufhin überprüft (vgl. [9], S. 18). Aus den Ergebnissen können Maßnahmen sowohl zur Fehlervermeidung als auch zur Fehlerbehandlung abgeleitet werden.

Die Validierung kann entweder durch Ausführung eines real existierenden Systems oder durch Simulation durchgeführt werden. Ersteres ist für funktionsfähige Prototypen oder sogar fertige Produkte interessant. Bei der simulationsbasierten Analyse wird ein Modell des Systems entworfen. In dieses Modell werden Fehler (*faults*) eingebaut und die Reaktion auf diese durch Simulation überprüft. Mit Hilfe der Fault-Injection können Soft- und Hardwarefehler gefunden werden. Bei Letzteren kann dies beispielsweise durch die Manipulation von Pins oder anderer externer Beschaltung vollzogen werden. Softwarefehler können ebenfalls sowohl in Simulationen als auch in real laufende Programme eingeschleust werden. Der Abstraktionsgrad reicht dabei von CPU-Registern bis hin zu ganzen Netzwerkpaketen in verteilten Systemen.

Erkennung und Behebung von Fehlern

Eine große Herausforderung beim Entwurf von verlässlichen Computersystemen stellt die Erkennung von Fehlern dar. In Echtzeitumgebungen muss dabei zusätzlich auf deren zeitlichen Umfang geachtet werden. Kopetz (vgl. [2], S. 220) beschreibt drei Vorgehensweisen um Fehler, zum Einen auf der funktionalen und zum Anderen auf der zeitlichen Ebene, zu erfassen.

Als Erstes sollte die Ausführungszeit eines Tasks überwacht werden. Falls dieser nicht innerhalb des zuvor spezifizierten Zeitrahmens abgearbeitet ist, wird dessen Ausführung vom Betriebssystem unterbrochen. Die Applikation kann nun entscheiden, was im Fehlerfall unternommen wird. Die hierbei zugrunde liegende Idee, die Ausführungszeit zu begrenzen, sollte in allen Teilen des Betriebssystems verwendet werden. Durch kontrollierte Timeouts lässt sich somit der Ausfall einer Komponente effizient feststellen. In diesem Zusammenhang muss allerdings darauf hingewiesen werden, dass mit dieser Methode ausschließlich sogenannte *Fail-Stop*-Ausfälle erkannt werden können. Byzantinische Ausfälle dagegen können nur über Konsensalgorithmen entdeckt und behoben werden (vgl. [10])

In ereignisgesteuerten Systemen ist weiter die Überwachung der Interrupteingänge nötig. Daher sollte das Betriebssystem zur Laufzeit die Zwischenankunftszeit von IRQs begrenzen können. Dies kann z. B. durch vorübergehende Deaktivierung erreicht werden.

Eine weitere Technik, die auch die Wertedomäne behandelt, ist die regelmäßige Ausführung von sogenannten *Challenge-Response* Protokollen. Diese sind ursprünglich aus der IT-Sicherheit bekannt, können, in leicht abgewandelter Form, aber auch in verlässlichen Systemen eingesetzt werden. Ein externer Controller stellt hierfür, der zu überwachenden Komponente, eine Eingabe zur Verfügung und erwartet innerhalb einer bestimmten Zeit eine zuvor festgelegte Antwort. Die Berechnung dieser sollte dabei möglichst viele funktionale Einheiten der Komponente involvieren. Ein einfaches Beispiel für ein Challenge-Response-Verfahren ist ein Watchdog-Timer. Die Applikation muss dabei periodisch ein Signal generieren (*heart beat*). Bleibt dieses für eine bestimmte Zeit aus, so wurde ein Fehler erkannt.

Erst nachdem ein Fehler erkannt wurde, können vom Betriebssystem Mechanismen zur Behebung desselben eingeleitet werden. In verlässlichen Computersystemen wird hierzu häufig eine Form der Redundanz genutzt[1]. Abhängig davon, ob das System Fehler in der Zeit- und/oder Wertedomäne maskieren soll, kann zwischen mehreren Arten unterschieden werden. Kopetz beschreibt dazu weiter die doppelte Ausführung von Tasks als eine effektive Methode zur Detektion von transienten Hardwarefehlern. In diesem Zusammenhang wird meist davon ausgegangen, dass die betroffene Software im Fehlerfall einfach neu gestartet wird. Dabei bleibt festzuhalten, dass ein Neustart bei Fehlern, in der Software selbst, das Problem häufig nicht beheben kann. Zusätzlich besteht die Gefahr, dass bspw. ein neu gestarteter Server auch alle Statusinformationen verliert. Aktive Verbindungen sind somit unterbrochen und müssen neu aufgebaut werden. Selbst wenn dies kein logisches Problem darstellt, so belastet es doch den zeitlichen Rahmen.

David et al. beschreibt in [11] ein interessantes Konzept, wie die Konsistenz zustandsbehafteter Informationen auch nach dem Neustart einer Komponente sichergestellt werden kann.

[1] bei byzantinischen Ausfällen wird Redundanz auch zur Erkennung von Fehlern eingesetzt

Fehlerisolation

Tanenbaum et al. ([12], S. 1) bezeichnet unzureichende oder gar fehlende Isolationsmechanismen als auch die Größe von aktuellen Betriebssystemen als Hauptursache für deren Unzuverlässigkeit. Da Fehler nur schwer komplett ausgeschlossen werden können, sollte zumindest darauf geachtet werden, dass sich diese nicht über die Grenzen einer Komponente verbreiten und dadurch das gesamte System kompromittieren. Mehrere Forschungsgruppen versuchen auf unterschiedliche Art und Weise dieses Problem zu beheben. Dabei lassen sich die populärsten Ansätze in vier Gruppen aufspalten:

Sandboxing Dabei wird potenziell gefährlicher Code (z. B. Gerätetreiber) in einer Art Schutzdomäne (*lightweight protection domain*) ausgeführt. Das zu schützende System ist dabei von einer Hülle (*wrapper*) umgeben, welche sämtliche Operationen der Komponente innerhalb der Schutzdomäne überwacht. Dadurch können Fehler erkannt und deren Verbreitung unterbunden werden.

Virtualisierung Anstatt ein gewöhnliches Betriebssystem zu nutzen, läuft hierbei ein spezieller *Virtueller Maschinen Monitor* (VMM) oder auch *Hypervisor* auf der Computerhardware. Dieser erlaubt es, mehrere virtuelle Instanzen der Hardware zu erstellen und diese, jeweils voneinander getrennt, ablaufen zu lassen. Die virtuelle Maschine (VM) kann dabei jede Software ausführen, die auch auf echter Hardware laufen würde (volle Virtualisierung). Müssen bestimmte Teile der Software angepasst werden, um in der virtuellen Umgebung laufen zu können, so spricht man von Paravirtualisierung. Xen [13] ist bspw. ein VMM der es gestattet, mehrere virtuelle Maschinen auf einer physischen ablaufen zu lassen. Auch VMware [14] oder VirtualBox [15] sind bekannte Virtualisierungslösungen.

Multi-Server-Betriebssystem Der Begriff beschreibt eine Klasse von Systemen, welche die Idee verfolgen, einen Großteil der Betriebssystemfunktionalität außerhalb des Systemkerns als Serverprozess zu implementieren. So existieren jeweils einzelne Server für den Zugriff auf Dateisysteme oder zur Nutzung des Druckers. Anwendungen (Clients) kommunizieren mit diesen über wohldefinierte Schnittstellen. Aufgrund des minimalen Funktionsumfangs wird der darunter liegende Kern häufig auch als Mikro- oder Nanokern bezeichnet. *Mach* [16] war

eine der ersten Implementierungen des Mikrokernkonzepts, dessen Entwicklung aber im Jahr 1994 eingestellt wurde. Aktiv gepflegte Vertreter dieser Gruppe sind *Minix3* [17] und Betriebssysteme, die auf der L4-Mikrokern-Familie aufbauen [18].

Sprachenbasierter Schutz Viele Probleme in Computersystemen werden durch Konstrukte und Artefakte systemnaher Programmiersprachen wie C oder C++ verursacht. Mit Hilfe von typsicheren Sprachen oder solchen, die bspw. das Pointerkonzept nicht vollständig unterstützen (keine Pointer-Arithmetik), kann die Erzeugung von potenziell gefährlichem Code eingeschränkt werden. Im Betriebssystem *Singularity* [19] kann so bspw. schon vom verwendeten Compiler überprüft und unterbunden werden, dass ein Prozess die Daten eines anderen überschreibt. Die Programmiersprache Java kann ebenfalls dieser Klasse zugeordnet werden.

2.3 Mikrokerne, L4/Fiasco, DROPS

Lange Zeit galt das Konzept der Mikrokerne, aufgrund der schlechten Performanz der ersten Implementierungen, als gescheitert. Anfang der 1990er revidierte Jochen Liedtke diese Annahme und zeigte, zunächst mit L3 und später mit L4, dass es mit Mikrokernen der zweiten Generation sehr wohl möglich ist, hoch performante Systemarchitekturen aufzubauen. Gegenüber dem ersten Mikrokern, Mach [16], stand neben der Verbesserung der Kommunikationspfade hauptsächlich eine Reduzierung der Komponenten im Vordergrund, die im Systemkern implementiert sind:

> *„A concept is tolerated inside the microkernel only if moving it outside the kernel, i.e., permitting competing implementations, would prevent the implementation of the system's required functionality."* ([20], S. 2)

L4-Mikrokerne unterstützen nur drei grundlegende Konzepte: Adressräume, Scheduling und Interprozesskommunikation (IPC). Sie bilden die Basis für alle weiteren Betriebssystemkomponenten, die darauf aufbauend, im Userspace implementiert werden. Dies schließt auch Subsysteme wie Speicher- und Interruptverwaltung, Netzwerkunterstützung, Dateisysteme und Gerätetreiber mit ein. Die Kombination aus L4-Mikrokern und allen Servicekomponenten bildet schließlich das Betriebssystem. Ziel der Architektur ist eine erhöhte Systemsicherheit- und Robustheit, Flexibilität sowie Modularität.

In monolithischen Betriebssystemen hingegen sind alle Teile in einer Komponente vereint. Der gesamte Programmcode läuft dort somit stets im privilegierten Modus. Abbildung 2.5 verdeutlicht die Reduzierung der privilegierten Komponenten (Kern-Modus) in mikrokernbasierten Architekturen, im Vergleich zu monolithischen Betriebssystemen.

Aufbauend auf dem ausschließlich auf x86-Systemen laufenden und in Assembler programmierten Mikrokern (L4/x86) von Jochen Liedtke, wurde ab 1998 an der Technischen Universität Dresden ein in C++ programmierter, echtzeitfähiger L4-Kern namens *Fiasco* [22] entwickelt. Dieser bildet die Basis für *DROPS* (*Dresden Real-time OPerating System*), einem Betriebssystem, welches neben nicht-echtzeitfähigen Anwendungen auch solche mit strengen zeitlichen Anforderungen unterstützt.

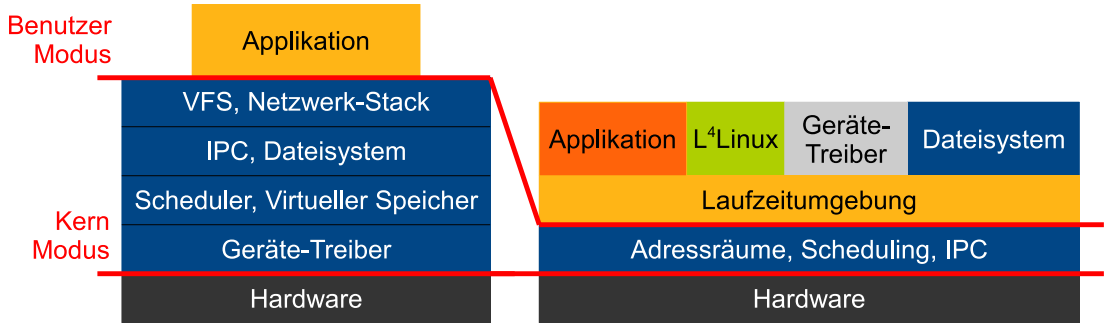


Abb. 2.5: Privilegierte Komponenten in monolithischen und mikrokernbasierten Betriebssystemarchitekturen (vgl. [21])

Eine Vielzahl von zusätzlichen Serverkomponenten, wie bspw. Speichermanager (*sigma0*), Interruptverwaltung (*omega0*) und I/O-Manager (*L4io*) bilden gemeinsam die obere Schicht des Betriebssystems. Die Laufzeitumgebung (*L4env*) abstrahiert von Mikrokernprimitiven, stellt eine Programmierschnittstelle zur Verfügung und unterstützt bzw. vereinfacht somit die Entwicklung von Anwendungsprogrammen in DROPS.

Eine weitere Funktionalität des Systems ist die Möglichkeit der Virtualisierung. Der Mikrokern selbst übernimmt dabei die Rolle eines minimalen *Hypervisors*. Er ermöglicht, selbst große, bereits vorhandene Systeme (*legacy system*) im unprivilegierten Userspace ablaufen zu lassen. L^{4}Linux beispielsweise ist eine paravirtualisierte Portierung des Linux-Kernels auf L4-Systemschnittstellen.

Eine weitere Möglichkeit zur Nutzung von Legacy-Komponenten in DROPS bietet das *Device Driver Environment* (DDE). DDE wurde ursprünglich mit der Idee entwickelt, die Vielzahl an vorhandenen Gerätetreibern (z. B. im Linux-Kern) auf L4-Systemen wiederzuverwenden. Hierbei wird allerdings nicht der gesamte Kern virtualisiert, vielmehr bildet DDE eine Linux-Umgebung auf das L4-Programmierinterface (*application programming interface*, *API*) ab. Dadurch wird die Kapselung einzelner Komponenten (z. B. Gerätetreiber oder sogar Netzwerkstacks) ermöglicht. In der in dieser Arbeit verwendeten Ausführung basiert das *Device Driver Environment* auf der API von Linux Version 2.6.29.

In DROPS kann somit eine Mischform aus Virtualisierung und Multi-Server-Konzept (vgl. Abschnitt 2.2) zur Fehlerisolation benutzt werden.

2.4 EtherCAT

EtherCAT [23] ist ein offenes, auf Ethernet-Hardware aufbauendes, echtzeitfähiges Kommunikationssystem. Es wurde mit dem Ziel entwickelt, einen kostengünstigen Feldbus mit kurzen Zykluszeiten und geringem Kommunikationsjitter zu schaffen. Typische Netzwerke erreichen beispielsweise Frequenzen zwischen $1kHz$ bis $20kHz$ (Zykluszeit $1ms$ bis $50\mu s$).

In der Regel ist EtherCAT als Single-Master-Bussystem konzipiert. Ein Algorithmus zur Kollisionserkennung ist nicht erforderlich, da der Masterknoten stets volle Kontrolle über den Bus hat. EtherCAT zeichnet sich weiter durch sehr kurze Verzögerungszeiten bei der Verarbeitung in den Slaveknoten aus. Dabei werden spezielle Kommunikationscontroller eingesetzt. Diese sind meist als FPGA (*field-programmable gate array*) oder ASIC *(application-specific integrated circuit)* implementiert und verarbeiten die EtherCAT-Pakete *on-the-fly* während diese den Knoten durchlaufen. Dabei werden die Pakete typischerweise nur um wenige Nanosekunden verzögert.

Der Master wird meist mit Hilfe von Standardkomponenten implementiert. Das Spektrum reicht dabei von handelsüblicher PC-Hardware mit Fast-Ethernet-Karte bis zu speziellen *System-On-a-Chip*-Rechnern mit integrierter Netzwerkhardware. Softwareseitig kommt hierbei ein sogenannter *Master-Stack* zum Einsatz. Dieser stellt, ähnlich wie ein TCP/IP-Stack, Anwendungen eine einheitliche Schnittstelle zur Verfügung und regelt die Kommunikation mit den darunter liegenden Schichten. Nachdem das System konfiguriert wurde, lässt sich die typische Interaktion zwischen Anwendungsprogramm und EtherCAT-Master vereinfacht in drei Schritte unterteilen: Prozessdaten lesen, Prozessdaten verarbeiten und Prozessdaten wieder aussenden.

Es existieren mindestens ein Dutzend kommerzielle und frei verfügbare Master-Implementierungen für verschiedene Betriebssystem- und Hardwarearchitekturen. Im Rahmen des EtherLab-Projektes [24] entstand ein quelloffener EtherCAT-Masterstack für Linux. Dieser befindet sich noch in der Entwicklung, der Quellcode zeichnet sich aber bereits jetzt durch hohe Qualität und Modularität aus. Die EtherCAT-Applikation wird hier, ebenso wie der gesamte Stack, als Modul in den Linux-Kernel eingebunden. Aktuell (Stand November 2009) befindet sich eine Userspace-Bibliothek in der Entwicklung, die es erlaubt, EtherCAT-Funktionalität in Anwendungen auch außerhalb des Systemkerns zu nutzen.

Kapitel 3

3 Verwandte Arbeiten

Im vorliegenden Kapitel soll eine Auswahl an existierenden Techniken bzw. Ansätzen zum Aufbau verlässlicher (Echtzeit-)Systeme präsentiert werden. Diese dienen als Einstiegspunkt in die Arbeit und liefern somit eine gute Basis für die Weiterentwicklung von Ideen.

Bei der Entwicklung verlässlicher Computersysteme ist Redundanz ein weitverbreiteter Lösungsansatz. Dabei laufen z. B. mehrere Instanzen einer Anwendung parallel nebeneinander. Ein Entscheider (*voter*) vergleicht deren individuell durchgeführte Berechnung und bildet daraus per Mehrheitsentscheid ein gemeinsames Ergebnis.

Engel et al. greift diese Idee in [25] auf. Er nutzt dabei ebenfalls DROPS, um z. B. zuverlässige Webserver zu implementieren. Mehrere Server laufen dabei parallel in getrennten L^{4}Linux-Instanzen. Zentraler Baustein dieser Architektur ist der so genannte Paket-Multiplexer. Dessen Aufgabe ist es, Netzwerk-Pakete zwischen den Webservern, diese haben nur Zugriff auf ein virtuelles Netzwerk, und einer weiteren L^{4}Linux-Instanz mit physischem Netzzugang zu vermitteln. Der Paket-Multiplexer ist es auch, der die Antworten der verschiedenen Server miteinander vergleicht und per Mehrheitsentscheid versucht, fehlerhafte Berechnungen zu erkennen. Diese Idee wird in der vorliegenden Arbeit mit dem Ziel aufgegriffen, Fehler bzw. Ausfälle in Gerätetreibern zu erkennen bzw. zu beheben. Im Kontext von Netzwerktreibern ist es jedoch nicht sinnvoll, mehrere Instanzen parallel ablaufen zu lassen. Sie würden sich beim Zugriff auf das Kommunikationsmedium behindern, sodass an dieser Stelle ein anderer Lösungsweg gefunden werden muss.

Einen etwas anderen Ansatz verfolgt Swift et al. in [26]. Ziel ist es, besonders Userspace-Anwendungen beim Ausfall von Gerätetreibern zu schützen. Dabei werden sogenannte Schattentreiber (*shadow drivers*) als Schicht zwischen Anwendung und nativen Gerätetreibern eingesetzt. Der Schattentreiber verhält sich gegenüber der Anwendung wie ein gewöhnlicher Treiber, fungiert also als Proxy. Hauptaufgabe dabei ist die Zwischenpufferung der Kommunikation. Fällt der native Treiber aus, so bricht nicht gleich die Verbindung zur Anwendung (und damit die gesamte Kommunikation) zusammen. Auch diese Idee soll in der vorliegenden Arbeit aufgenommen und für die Verwirklichung von verlässlichen EtherCAT-Master-Knoten weiterentwickelt werden.

Kapitel 4

4 Echtzeit- und Verlässlichkeitsanalyse

Das vorliegende Kapitel befasst sich mit der theoretischen Analyse von monolithischen- und mikrokernbasierten Betriebssystemarchitekturen. Am Beispiel von zwei Vertretern beider Architekturkonzepte, Linux und DROPS, soll deren grundsätzliche Eignung für den Einsatz in verlässlichen und echtzeitfähigen Computersystemen untersucht werden.

In Abschnitt 4.1 werden zunächst Echtzeiteigenschaften beider Beispielsysteme diskutiert. Im Anschluss daran erfolgt im Abschnitt 4.2 die Analyse hinsichtlich deren Verlässlichkeit. Abschnitt 4.3 fasst die gewonnenen Erkenntnisse zusammen, diskutiert Wechselwirkungen und setzt diese in einen gemeinsamen Kontext.

4.1 Echtzeitanalyse

Ziel dieses Kapitels ist der Vergleich von monolithischen- und mikrokernbasierten Betriebssystemarchitekturen bezüglich deren Echtzeiteigenschaften. Zu diesem Zweck werden im Folgenden die wichtigsten Betriebssystemkomponenten untersucht. Die Betrachtungen beziehen sich dabei auf zwei konkrete Implementierungen beider Konzepte. Es soll angemerkt werden, dass jeder der folgenden Abschnitte in drei Teile: Grundlagen, Diskussion und Fazit untergliedert ist. Der Abschnitt Diskussion

beschäftigt sich dabei jeweils im ersten Absatz mit Eigenschaften des Linux-Systems und thematisiert schließlich im zweiten Absatz DROPS.

Die Analyse als auch die Evaluation finden auf Basis derselben Hard- und Softwarekonfiguration statt. Das System wird daher detailliert im Kapitel 6.1 vorgestellt.

4.1.1 Scheduling

Dieser Abschnitt vergleicht Echtzeitaspekte beim Scheduling in monolithischen- und mikrokernbasierten Betriebssystemen am Beispiel von Linux und DROPS. Dafür wird zunächst das Schedulingproblem erläutert und dieses in Zusammenhang mit Echtzeitsystemen gebracht. Des Weiteren werden Faktoren erörtert, welche die Erfüllung zeitlicher Anforderungen beim Scheduling gefährden.

Grundlagen

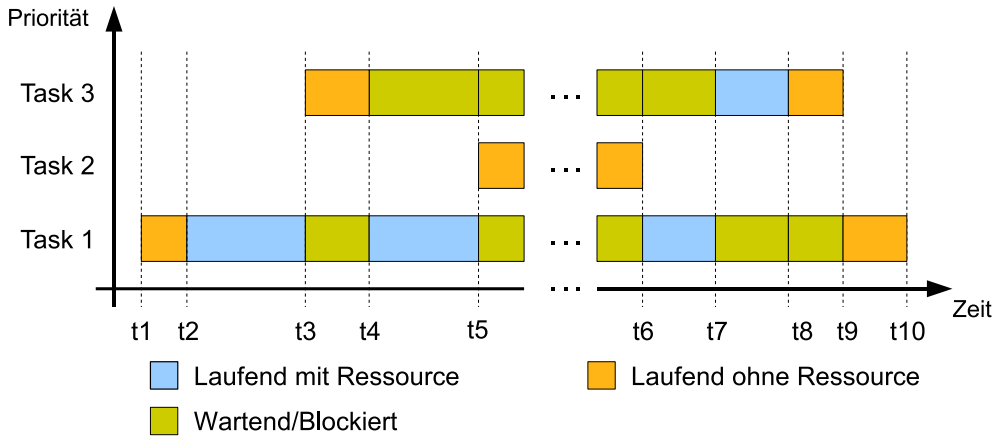
Als Scheduling bezeichnet man in der Informatik allgemein die Aufteilung (zeitlich) begrenzter Ressourcen auf mehrere konkurrierende Instanzen. Das Spektrum reicht dabei von CPU-Scheduling über I/O-Scheduling bis hin zu Netzwerk-Scheduling. In dieser Arbeit bezieht sich der Begriff Scheduling jedoch ausschließlich auf die Ressource Prozessor und beschreibt somit die Frage, welcher Task oder Thread zu einer bestimmten Zeit auf der CPU abläuft. Die Definition von Prozess, Task und Thread weicht dabei in unterschiedlichen Systemen leicht voneinander ab. Als Task bezeichnet man allgemein die Umgebung zum Ablauf eines Computerprogramms. Hierzu zählt zum Beispiel ein Adressraum, Speicher sowie andere Betriebsmittel. Innerhalb dieser Umgebung existiert nun ein oder auch mehrere Thread(s). Diese sind die Aktivitätsträger und führen den jeweiligen Programmcode aus. Sie besitzen jeweils einen eigenen Programmzähler, einen eigenen Stapel sowie eine bestimmte Anzahl an Registern, auf denen sie operieren können. Unter Linux wird ein Task auch als Prozess bezeichnet. Diese zwei Begriffe können somit als Synonym verwendet werden.

Die Auswahl, welcher Thread als Nächster laufen soll, kann entweder auf Basis eines zuvor festgelegten Zeitplans geschehen (*Offline-Scheduling*) oder durch einen zur Laufzeit ausgeführten Algorithmus entschieden werden (*Online-Scheduling*). Die

Betriebssystemkomponente, die diese Entscheidung trifft, nennt man Scheduler. Bei Online-Schedulingverfahren sind Scheduler und Schedulingalgorithmus oft in einer Komponente vereint. Man spricht hierbei einfach von Scheduler, meint aber die Kombination beider. Bei Offline-Schedulingverfahren wird im System meist nur ein *Dispatcher* implementiert, der dann zur Laufzeit den zuvor erstellten Zeitplan durchsetzt. Diese kommen bspw. in *OSEKtime* [27] zum Einsatz. Dabei muss jedoch sichergestellt werden, dass *a priori* alle Prozesse und benötigten Ressourcen statisch definiert wurden.

Schedulingverfahren werden weiter in *kooperativ* oder *präemptiv* unterteilt (vgl. [28], S. 152). Nicht unterbrechende Scheduler lassen einen Task solange laufen, bis dieser die CPU selbst freigibt oder aber, auf eine Ressource wartend, blockiert. In Echtzeitsystemen ist dieses Verfahren jedoch nicht gebräuchlich, da in nicht-trivialen Systemen keine Aussage über die Laufzeit und das (möglicherweise böswillige) Verhalten der einzelnen Tasks getroffen werden kann. Somit stellt sich die Frage, welcher Algorithmus der richtige ist. Entscheidend hierfür sind die angestrebten Systemziele. In Echtzeitcomputern ist dies die garantierte Erfüllung von Fristen. Ein Großteil der heute in der Praxis eingesetzten Verfahren unterstützt allerdings nicht die Angabe von Schedulingparametern wie der WCET, der Frist oder der Zykluszeit. Häufig werden diese ausschließlich über Prioritäten beschrieben, also nur eine Gruppierung einzelner Prozesse anhand ihrer Wichtigkeit vorgenommen. Die Ausdruckskraft einmal festgelegter Prioritäten ist aber stark begrenzt, sodass aktuell (zum Beispiel in Linux) wieder ein starker Trend bei der Entwicklung und Erprobung von Verfahren wie EDF (*earliest deadline first*) zu beobachten ist.

Allen Verfahren ist schließlich die Herausforderung gemein, eine Menge von konkurrierenden Tasks so auszuführen, dass alle zeitkritischen Tasks in der Lage sind, ihren zuvor spezifizierten Zeitrahmen zu erfüllen. In der Praxis gefährden die unterschiedlichsten Aspekte die Erfüllung dieses Ziels. Beispielsweise können sporadisch eintreffende Interrupts die Abarbeitung wichtiger Tasks stören. Treten diese Unterbrechungen nun vermehrt auf, so läuft das System Gefahr, seine Arbeit nicht rechtzeitig beenden zu können (vgl. Abschnitt 4.1.3).



Zeit	Task 1	Task 2	Task 3
t1	wird aktiviert und gestartet		
t2	belegt Ressource (R)		
t3	wird verdrängt		wird aktiviert und gestartet
t4	wird wieder gestartet		möchte R, wird verdrängt
t5		aktiviert und gestartet	
t6	wird wieder gestartet	beendet sich	
t7	gibt R frei, wird verdrängt		wird wieder gestartet, belegt R
t8			gibt R frei
t9	wird wieder gestartet		beendet sich
t10	beendet sich		

Abb. 4.1: Beispiel zur Prioritätsinversion (vgl. [29])

Konkurrieren mehrere Tasks um eine gemeinsam genutzte Ressource, so kann weiterhin das Problem der Prioritätsinversion (*priority inversion*) entstehen. Dabei sind mindestens drei Tasks (T1, T2, T3) und eine Systemressource, die jeweils nur exklusiv genutzt werden kann, beteiligt. Angenommen, ein niedrig priorisierter Task (T1) hält gerade die Ressource (R). Möchte nun ein hochpriorer Task (T3) diese nutzen, so blockiert er aufgrund der exklusiven Nutzung durch T1. T3 wartet nun, bis Task 1 die Ressource nicht mehr benötigt und diese freigibt. Wird nun an dieser Stelle einem Task (T2) mit mittlerer Priorität die CPU zugeteilt, so verdrängt dieser T1 und dessen Abarbeitung verzögert sich. T3 wartet somit auf T2, die Prioritäten von Task 3 und Task 2 sind demnach vertauscht. Abbildung 4.1 veranschaulicht diese Situation.

Diskussion

Der Standardscheduler im Linux-Kern der Serie 2.6 hat sich im Vergleich zu Vorgängerversionen erheblich verbessert. Die Laufzeit des Algorithmus hängt zum Beispiel nicht mehr von der Prozessanzahl ab. Zusätzlich wurden eigene Schedulingklassen für Echtzeittasks eingeführt.

Intern arbeitet der Scheduler mit festen (aber frei wählbaren) Prioritäten. Dabei wird ein hochpriorisierter Thread immer bevorzugt behandelt und ist in der Lage, andere niedriger priorisierte Threads zu verdrängen. Man unterteilt Echtzeitthreads in eine von zwei Schedulingklassen: `SCHED_FIFO` oder `SCHED_RR`. `SCHED_FIFO` bedeutet dabei, dass einem laufenden Thread erst dann wieder die CPU entzogen wird, wenn dieser sie freiwillig aufgibt. Natürlich können höher priorisierte Threads den laufenden Thread verdrängen, innerhalb einer Priorität jedoch werden Threads der `SCHED_FIFO`-Klasse nicht unterbrochen.

`SCHED_RR` hingegen garantiert Fairness innerhalb derselben Priorität, ein Prozess läuft maximal solange, bis dessen Zeitscheibe aufgebraucht ist. Danach wird er an das Ende der *Expired*-Warteschlange gesetzt und der nächste Prozess der *Active*-Warteschlange aufgerufen. Warteschlangen sind spezielle Datenstrukturen, die (beliebig viele) Objekte nach dem FIFO-Prinzip[1] verwalten. Jeder Prozesspriorität sind zwei dieser Warteschlangen (die Active- und Expired-Warteschlange) zugeordnet. Somit wird kontinuierlich jede Prioritätsklasse abgearbeitet, bis am Ende kein Prozess mehr in der jeweiligen Active-Warteschlange vorhanden ist. Danach werden die Zeiger auf beide Warteschlangen vertauscht und der Scheduler beginnt erneut mit seiner Arbeit. Man beachte, dass dieses Konzept verhindert, dass niedrig-priorisierte Prozesse "verhungern" (*starvation*). Allerdings bietet es diesen auch die Gelegenheit, das zeitliche Verhalten von Echtzeitprozessen zu stören. Dabei bleibt jedoch anzumerken, dass Linux niedrigprioren Tasks automatisch kürzere Zeitscheiben als höher priorisierten Prozessen zuweist (vgl. [30], S. 754).

Um im Userspace Threads vor ungewollter Prioritätsinversion zu schützen, kann ein Anwendungsentwickler in Linux ab Kernel 2.6.18 Gebrauch von sogenannten *Futex* (*fast userspace mutex*) machen. Diese, auch als PI-Mutex bekannten Konstrukte, implementieren Locking mit Hilfe von Prioritätsvererbung.

[1] First-In-First-Out Prinzip

Unter DROPS ist dessen Systemkern Fiasco für das Scheduling verantwortlich. Anzumerken ist, dass sich diese Funktionalität streng genommen auch im Userspace implementieren ließe und somit gegen das „Minimalitätsprinzip" von mikrokernbasierten Systemen spricht (vgl. Kapitel 2.3).

Das Scheduling in Fiasco basiert ebenfalls auf statischen Thread-Prioritäten. Threads innerhalb einer Priorität werden nach einem zeitscheibenbasierten Round-Robin-Verfahren behandelt. Fiasco garantiert somit Fairness innerhalb einer Thread-Priorität. Die Länge der Zeitscheiben, auch Quanta genannt, ist frei wählbar. Das aus Linux bekannte FIFO-Scheduling kann emuliert werden, indem jedem Thread die gleiche Priorität und eine unendlich lange Zeitscheibe zugeordnet wird.

In Fiasco werden hauptsächlich nicht blockierende Techniken zur Synchronisation eingesetzt [31]. Nur sehr kurze Abschnitte sind weiterhin durch Sperren der Interrupts geschützt, der Kern ist somit fast vollständig unterbrechbar. Auf Applikationsebene können Threads jedoch weiterhin um gemeinsam genutzte Ressourcen konkurrieren. Die Gefahr der ungewollten Prioritätsinversion besteht somit weiterhin. Dieser wird in Fiasco mit einer Technik namens *Timeslice-Donation* entgegengewirkt (vgl. [32], Folie 39). Die grundlegende Idee dabei ist Kooperation. Ein hochpriorer Thread unterstützt die Abarbeitung eines niederprioren Threads indem dieser ihm seine Priorität und aktuelle Zeitscheibe vererbt (*priority inheritance*). Ein Server, der beispielsweise eine IPC-Botschaft von einem Client empfängt und daraufhin die Bearbeitung einer Aufgabe startet, verbraucht somit CPU-Zeit des Clients. Timeslice-Donation wird durch die Aufspaltung des TCB (*thread control block*) in Ausführungskontext (*execution context)* und Schedulingkontext (*scheduling context*) ermöglicht. Ersterer beinhaltet nur den CPU-Status, also z. B. die Registerzustände. Der Schedulingkontext hingegen speichert Zeitscheiben- und Prioritätsinformationen. Bei einem Threadwechsel, der durch IPC ausgelöst wurde, ändert sich demnach nur der Ausführungskontext.

Fazit

Scheduling unterscheidet sich bis auf Details in beiden hier betrachteten Systemen nur geringfügig voneinander - beide nutzen präemptives, prioritätsbasiertes (Round-Robin-)Scheduling. Prioritätsinversion wird in beiden betrachteten Systemen durch Prioritätsvererbung verhindert.

4.1.2 Kommunikation

Ziel dieses Abschnitts ist die grundlegende Beschreibung der Interprozesskommunikation (*IPC*) in monolithischen und mikrokernbasierten Architekturen am Beispiel von Linux und DROPS. Schwerpunktmäßig wird die Tauglichkeit zur Erfüllung von Echtzeitanforderungen untersucht.

Grundlagen

Das Anwendungsspektrum von Interprozesskommunikation in Computersystemen ist vielseitig und beschränkt sich nicht ausschließlich auf den Austausch von Daten. Ebenso wichtig ist bspw. auch die Synchronisation von parallel laufenden Threads oder Prozessen. Speziell in L4-basierten Systemen kommt zusätzlich die Behandlung von Interrupts (siehe Abschnitt 4.1.3) sowie der Zugriff auf Speicher oder I/O-Ports hinzu.

Echtzeitsysteme beinhalten oft auch zeitunkritische Teilkomponenten. Diese dürfen selbstverständlich die Ausführung der Echtzeittasks nicht stören. Es soll aber dennoch ermöglicht werden, Daten auch zwischen zeitkritischen und weniger zeitkritischen Applikationen auszutauschen. Für diesen Zweck besonders interessant ist die gemeinsame Nutzung von Speicher (*shared memory*). Dabei werden die gleichen physischen Speicherseiten in zwei oder mehrere virtuelle Adressräume eingeblendet. Auf Basis von Shared-Memory lassen sich effiziente Datenstrukturen zum Austausch von Informationen zwischen echtzeitfähigen und nichtechtzeitfähigen Prozessen, beispielsweise einem Steuer- und Überwachungsprogramm, aufbauen. Diese Art der Kommunikation ist besonders effizient, da aufwendiges Kopieren der Daten entfällt. Beim Zugriff auf die Speicherbereiche ist jedoch stets auf eine geeignete Form der Synchronisation zu achten.

Diskussion

In Linux-Systemen existiert eine Vielzahl von Varianten zur Interprozesskommunikation. Die Behandlung aller möglichen Techniken ist im Rahmen dieser Arbeit nicht vorgesehen, detaillierte Informationen hierzu finden sich aber beispielsweise in [33].

Wie im oberen Abschnitt bereits erwähnt, eignet sich Shared-Memory gut für die Verwendung in Echtzeitsystemen, besonders wenn das betroffene Kommunikationsszenario dies begünstigt, also bspw. große Datenmengen transportiert werden müssen (z. B. in *Streamingapplikationen*).

Unter Linux kann hierzu eine Familie von Funktionen rund um `shmget()` verwendet werden. Fertige Bibliotheken, ähnlich denen in *RTAI* (*RealTime Application Interface for Linux*) [34], existieren nicht. Der Benutzer muss sich also selbstständig um die Synchronisation kümmern. Dabei können beispielsweise *Futexe* zum Einsatz kommen (vgl. Abschnitt 4.1.1).

Der überwiegende Teil der Betriebssystemfunktionalität ist in mikrokernbasierten Architekturen in getrennten Komponenten implementiert. Nutzer und Anbieter von Systemservices müssen daher häufig über Adressraumgrenzen hinweg miteinander kommunizieren. Effiziente Kommunikationsmechanismen sind aus diesem Grund besonders wichtig.

L4-basierte Mikrokerne benutzen zum Nachrichtenaustausch synchrone IPC. Zwei Partner müssen hierbei der Kommunikation explizit zustimmen, erst dann werden Daten vom Sender zum Empfänger kopiert. Hierdurch werden doppelte Kopien oder die Pufferung der Daten im Systemkern vermieden. Die effizienteste Form des Nachrichtenaustauschs stellt die sogenannte *Short-IPC* oder *Register-IPC* dar. Dabei werden die Daten nicht über den Arbeitsspeicher übertragen. Vielmehr verbleiben die Informationen in bestimmten Prozessorregistern und können nach dem Kontextwechsel vom Empfänger sofort gelesen werden. Nachteil dieser Methode ist allerdings die begrenzte Nachrichtengröße: Es können nur zwei Datenworte übertragen werden. Sollen wesentlich mehr Informationen transportiert werden, so kommt entweder *Long-IPC* (auch *Memory-IPC* genannt) oder gemeinsam genutzter Speicher zum Einsatz. Mit Long-IPC können bis zu 524.288 (2^{19}) Doppelworte übertragen werden (vgl. [35]). Diese Methode hat allerdings den Nachteil, dass Seitenfehler und dadurch Verzögerungen auftreten können. In Echtzeitsystemen wird aus diesem Grund häufig vollkommen auf Long-IPC verzichtet und nur Short-IPC und Shared-Memory genutzt. Letzterer wird in DROPS direkt vom *Dataspace-Manager* bereitgestellt. Nutzer müssen sich dabei allerdings, ähnlich wie bei Linux, selbst um die Zugriffssynchronisation kümmern. Mit *DSI* (*DROPS Streaming Interface*) steht dem Anwender

allerdings auch ein paketorientiertes Transportprotokoll zur Verfügung, welches sogar jitterbegrenzte periodische Kommunikation zulässt (vgl. [36]).

Alle IPC-Ausprägungen in L4 können mit Timeouts versehen werden. Dieser Tatsache ist es zu verdanken, dass von sämtlichen Betriebssystemfunktionen die maximale Ausführungszeit begrenzt werden kann. Die Zeiten können dabei von Null (nicht-blockierende IPC) bis zu unendlich (solange blockieren, bis Partner Ergebnis liefert) variieren. Die genaue Auflösung der Wartezeit hängt von der verwendeten Zeitbasis ab.

Fazit

Das Kommunizieren von Daten ist eine der Hauptaufgaben heutiger Computersysteme. Dieser Abschnitt hat Methoden und Konzepte zum Nachrichtenaustausch in zwei unterschiedlichen Betriebssystemen erörtert. Obwohl sich Voraussetzungen sowie konkrete Implementierungen in beiden Architekturen stark voneinander abheben, findet der Anwender dennoch ähnliche Technologien, um Echtzeitapplikationen zu realisieren. Sollen Informationen innerhalb, aber auch über Adressraumgrenzen hinweg ausgetauscht werden, so kann, unabhängig von der Betriebssystemarchitektur, gemeinsam genutzter Speicher zum Einsatz kommen. Um deterministisches Verhalten zu gewährleisten, sollten dabei jedoch nicht-blockierende Synchronisationstechniken genutzt werden. Die Möglichkeit, unter L4 Timeouts für den Nachrichtenaustausch festzulegen, ist in zeitkritischen Systemen von großem Vorteil.

4.1.3 Interruptverwaltung

Interrupts nehmen in Echtzeitsystemen eine besondere Stellung ein. Die Behandlung dieser ist somit ein essenzieller Bestandteil der Betriebssystemfunktionalität (vgl. [37], S. 504). Der erste Teil dieses Abschnitts beschreibt daher allgemeingültige Grundlagen der Interrupt-Behandlung in Echtzeitsystemen. Zusätzlich werden *Best-Practice-Strategien* bei der Umsetzung in Betriebssystemen gezeigt. Der zweite Teil widmet sich der tatsächlichen Implementierung in Linux und DROPS.

Grundlagen

Zeitverzögerungen bei der Behandlung von Interrupts können zu fehlerhaftem Verhalten zeitkritischer Anwendungen führen: Beispielsweise können Latenzen bei der Audio- und/oder Videoübertragung entstehen oder kritische Sensordaten verloren gehen. Andererseits können zu viele Interrupts das Systemverhalten auch empfindlich stören.

Interrupt-Requests (IRQs) werden z. B. von Timern entweder periodisch oder einmalig ausgelöst. Dabei markieren diese Zeitabschnitte oder lösen etwa das Starten oder Stoppen von systemrelevanten Operationen aus. So wird beispielsweise der periodische Aufruf des Schedulers (vgl. Abschnitt 4.1.1 und 4.1.5) durch Timer-Interrupts angestoßen.

IRQs werden jedoch auch von externer Hardware oder Peripherie abgesetzt. Dabei sind dies effiziente Mechanismen, um ein System über das Auftreten von externen, möglicherweise sporadischen Signalen bzw. Ereignissen in Kenntnis zu setzen.

In Abhängigkeit von der Interruptquelle kann die Bearbeitung der gesamten *Interrupt-Service-Routine* (ISR) unterschiedlich lang dauern. Die ISR eines Netzwerkgerätes bspw. implementiert in der Regel eine recht komplexe Funktionalität: Sobald ein neues Paket eintrifft, löst die Netzwerkhardware einen Interrupt aus. Der entsprechende Interrupt-Handler überprüft das Ereignis, fragt evtl. aufgetretene Fehler ab und kopiert schließlich die empfangenen Daten in einen, vom Netzwerksystem zur Verfügung gestellten, Zwischenspeicher. Die Abarbeitung solch komplexer Funktionen kann leicht mehrere Millisekunden beanspruchen und variiert möglicherweise in Abhängigkeit von der momentanen Last. Ein Interrupt kann bspw. mehrere eingetroffene Pakete signalisieren. Das Kopieren der Daten würde damit mehr Zeit in Anspruch nehmen und somit die Einhaltung der Fristen gefährden. Aus diesem Grund behandeln die meisten Echtzeitbetriebssysteme Interrupts in zwei unterschiedlichen Phasen: *First-Level Interrupt-Handler* (FLIH) und *Second-Level Interrupt-Handler* (SLIH):

First-Level Interrupt-Handler Die erste Phase, auch *Top-Half* oder *Hard Interrupt-Handler* genannt, wird nach Auftreten des Interrupts gestartet. Aufgabe dieser ISR ist es nun, so schnell wie möglich diesen, für den WCAO (vgl. Kapitel 2.1: Evaluation) kritischen Teil der Interrupt-Verarbeitung, zu beenden. Dabei

werden hauptsächlich Daten verarbeitet, die tatsächlich nur kurz nach Auftreten des Interrupts verfügbar sind. Alle anderen Aufgaben sollen später im SLIH durchgeführt werden.

Second-Level Interrupt-Handler Möglicherweise zeitaufwendige Aufgaben bei der Interrupt-Bearbeitung sollten in der zweiten Phase, auch *Bottom-Half* oder *Soft Interrupt-Handler* genannt, vollzogen werden. Diese Routine ist typischerweise komplett unterbrechbar und wird, abhängig von deren Priorität, vom Scheduler aufgerufen.

Im Folgenden werden Aspekte im Zusammenhang mit der Interruptverarbeitung erläutert, die in einem Echtzeitsystem eine Bedrohung für dessen korrekte Funktionalität darstellen (vgl. [38], S. 11).

Besonders kritisch sind Zeitspannen einzuschätzen, in denen Interrupts gänzlich ausgeschaltet sind. Hierdurch können Ereignisse nicht signalisiert, der periodische Scheduleraufruf verzögert oder, im schlimmsten Fall, gänzlich unterbunden werden. Ein ähnliches Problem stellen lang andauernde Top-Half Interrupt-Handler dar, da nach einem Kerneintritt durch einen Interrupt *(interruptgate)* ebenfalls alle Interrupts gesperrt sind[2]. Des Weiteren sollte in Echtzeitsystemen auf die korrekte Vergabe von Prioritäten an die entsprechenden IRQ-Handler geachtet werden. Dies soll verhindern, dass besonders kritische Interrupts nicht von weniger wichtigen Unterbrechungen verzögert werden.

Eine zusätzliche Gefahr für Echtzeitsysteme stellt eine Interrupt-Überlast-Situation dar. Hierbei wird, von einem möglicherweise kompromittierten System, mit hoher Frequenz eine Vielzahl von Unterbrechungsanforderungen generiert. Ein darauf nicht vorbereiteter Echtzeitcomputer könnte, da er zu einhundert Prozent mit der Bearbeitung von Interrupts belegt ist, keine weiteren Berechnungen vornehmen. Dadurch kann die Erfüllung zeitlicher Anforderungen nicht mehr gewährleistet werden.

[2] konkretes Verhalten ist architekturabhängig: In x86-Systemen sind alle Interrupts am PIC maskiert

Diskussion

Angenommen ein Rechensystem befindet sich gerade bei der Abarbeitung einer Anwendung im Userspace. Trifft nun ein Interrupt ein, so wechselt die Hardware durch diesen automatisch in den privilegierten Betriebsmodus. Der Systemkern ruft nun alle Top-Half Interrupt-Handler auf, die mit dem entsprechenden IRQ verbunden sind. Aus oben erwähnten Gründen ist in Echtzeitsystemen die Abarbeitung aufwendiger Routinen an dieser Stelle nicht empfehlenswert. Daher wird hier nur der Aufruf des Bottom-Half Interrupt-Handlers vorbereitet. Beim Beenden der Top-Half überprüft der Kernel, ob Soft-IRQs zur Bearbeitung anstehen. Ist dies der Fall, so kann bei entsprechend hoher Priorität deren Bearbeitung sofort ohne zusätzliche indirekte Kosten[3] gestartet werden. Da während der Interrupt-Bearbeitung neue IRQs eintreffen und diese wiederum neue Bottom-Half-Handler vorbereiten können, wird zusätzlich die maximale Aufrufanzahl begrenzt. Dadurch kann die Entstehung von Überlast durch den unendlichen Aufruf von Bottom-Half-Handlern verhindert werden. Untenstehender Algorithmus verdeutlicht die Interrupt-Behandlung im Linux-Kernel.

Algorithmus 1 Behandlung eines I/O-Interrupts im Linux Kern (Pseudocode)

```
1: save context of current process
2: for pending interrupt do
3:     call all top-half irq handler
4: end for
5: while bottom-half registered && !max_calls_reached do
6:     call bottom-half irq handler
7: end while
8: restore process context
```

Die Interruptbehandlung in DROPS unterscheidet sich im Vergleich zu Linux-basierten Systemen erheblich. Wie in allen L4-basierten Architekturen werden bis auf den Timer-Aufruf alle Interrupts in IPC-Nachrichten umgewandelt und an den entsprechenden Userspace-Handler weitergereicht. Bevor nun aber tatsächlich ein Treiber die gerätespezifischen Operationen ausführen kann, muss ein Interrupt in DROPS typischerweise eine weitere Zwischenschicht durchlaufen: *Omega0*. Dies ist

[3] Adressraum- oder Prioritätslevelwechsel

ein Userspace-Server zur Interruptverwaltung, dessen Aufgabe darin besteht, die gesamte Interruptlogik zu abstrahieren und Gerätetreibern ein architekturunabhängiges Interface zur Verfügung zu stellen. Grundsätzlich ist die Verwendung von Omega0 nicht zwingend notwendig, da sich Gerätetreiber auch direkt am Kern als Interrupt-Handler registrieren können. Vorteil beim Einsatz ist allerdings die Möglichkeit, dass mehrere Treiber den gleichen Interrupt nutzen können (*interrupt sharing*). Omega0 fungiert in diesem Fall als Vermittler und nimmt mit der zentralen Verwaltung der Interrupts auch eine zentrale Rolle in der Gesamtarchitektur ein. Daher sollte diese Komponente auch zur *TCB* (*trusted computing base*) des Betriebssystems gezählt werden (vgl. [39], S. 5). Abbildung 4.2 verdeutlicht die Interrupt-Architektur mit Omega0.

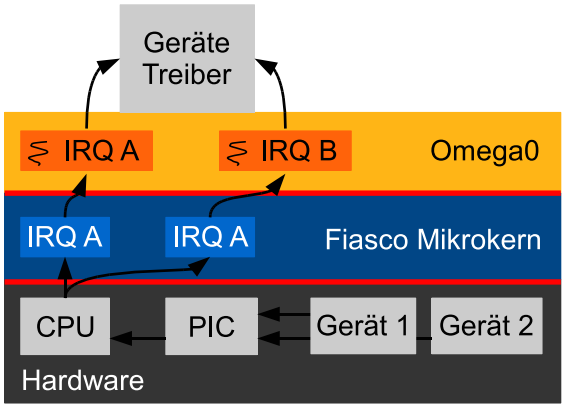


Abb. 4.2: Omega0 Architektur in DROPS (vgl. [40], Folie 16)

Im Folgenden soll nun die Interruptverarbeitung in DROPS erläutert werden. Sie beginnt mit der Generierung einer Unterbrechungsanforderung für die CPU. Wird diese erkannt, so beendet der Prozessor noch die gerade ausgeführte Instruktion. Anschließend findet ein Wechsel zum entsprechenden Kernel-Interrupt-Handler statt. Dessen Aufgabe ist die Bestätigung des IRQ am PIC-Controller sowie die Weiterleitung des Interrupts an den entsprechenden (Omega0-)Handler-Thread. Hat sich zuvor ein Gerätetreiber für den Interrupt registriert, so wird dieser nun von Omega0 an den entsprechenden treiberspezifischen Handler weitergereicht. In Bezug auf die Interruptlatenz wirkt sich diese zweischichtige Architektur nachteilig aus. Der zusätzliche Aufwand begründet sich hauptsächlich durch die größere Codekomplexität und die erhöhte Anzahl an Adressraumwechseln. Des Weiteren ist mindestens ein

Prioritätslevelwechsel nötig, falls sich die CPU gerade im Kernmodus befand. Zwei Wechsel sind hingegen nötig, falls sich die CPU zum Interruptzeitpunkt gerade im Usermodus befand.

Interrupt-Handler verhalten sich wie normale Threads, so können ihnen bspw. Prioritäten zugeordnet werden. Außerdem sind sie unterbrechbar und werden bei Bedarf einfach vom Scheduler verdrängt. Die Ausführung anderer, höher priorisierter Threads oder Interrupt-Handler kann somit gewährleistet werden. Die Verlagerung der Interruptbehandlung in den Userspace bietet weiterhin den Vorteil, dass Gerätetreiber nicht einfach systemweit alle Interrupts sperren können. Dieser Umstand steigert die Vorhersagbarkeit des Gesamtsystems erheblich.

Interrupt-Überlast wird in L4-basierten Architekturen inhärent dadurch verhindert, dass der Handler selbst darüber verfügt, wann der nächste Interrupt zugelassen wird.

Fazit

In diesem Abschnitt wurden Echtzeitaspekte bei der Interruptverarbeitung in Linux und DROPS diskutiert. Konzeptbedingt unterscheiden sich beide Implementierungen erheblich voneinander. Die zweistufige Architektur in DROPS ist etwas komplexer und erzeugt insgesamt mehr Overhead bei der Bearbeitung von Interrupts. Aus diesem Grund sind leicht höhere Verzögerungen, sowohl im schlechtesten als auch im besten Fall, zu erwarten. Die entstehenden Kosten sind jedoch in etwa mit denen vergleichbar, die durch blockierte Interrupts bzw. durch moderne Hardwarefeatures (z. B. Caches) erzeugt werden (vgl. [41], S. 9).

4.1.4 Speicherverwaltung

Neben der Rechenzeit ist Speicher eine der wichtigsten Ressourcen in Computersystemen. Für einen problemlosen Betrieb von Echtzeitrechnern sind in diesem Zusammenhang jedoch einige Grundregeln zu beachten. Diese sollen im vorliegenden Abschnitt ebenso wie der Einsatz von Prozessor- und MMU-Caches betrachtet werden.

Grundlagen

Heute bieten nahezu alle modernen Betriebssysteme virtuellen Speicher. Dieses Konzept bietet einem Programm die Illusion eines eigenen, zusammenhängenden Speicher- oder Adressbereichs. Tatsächlich kann dieser in mehrere unterschiedlich große Teile aufgeteilt sein und sich teilweise sogar auf verschiedenen Speicherträgern befinden.

Die Entwicklung von speicherintensiven Programmen wird in Systemen mit virtuellem Speicher wesentlich vereinfacht, außerdem kann der tatsächlich physisch vorhandene Speicher effizienter genutzt werden. Allerdings bringt der Einsatz auch einige Nachteile mit sich: Die Erfüllung zeitlicher Anforderungen in Echtzeitsystemen gestaltet sich beispielsweise komplexer. Zum Einen muss das System, falls dynamischer Speicher innerhalb der Echtzeitkomponente verwendet wird, bei dessen Allozierung darauf achten, dass diese auch tatsächlich Erfolg hat und zusätzlich nicht die zeitlichen Vorgaben überschreitet. Zum Anderen muss für den späteren Zugriff sichergestellt werden, dass dieser nicht durch das Speicherverwaltungssystem verzögert wird. Eine Seitenzugriffsverletzung (*page fault*) mit anschließender Festplattenaktivität kann beispielsweise leicht mehrere Millisekunden dauern.

In Computerarchitekturen mit virtuellem Speicher werden sogenannte MMUs eingesetzt, um zu einer gegebenen virtuellen Adresse die passende physische Adresse zu finden. Die korrekte Zuordnung findet die MMU in der Seitentabelle (*page table*), die meist im Hauptspeicher gehalten wird. Damit die MMU nicht bei jeder Anfrage zunächst auf den verhältnismäßig langsamen Arbeitsspeicher zugreifen muss, ist dieser ein Cache, der sogenannte *TLB* (*translation lookaside buffer*), vorgeschaltet. Der TLB puffert allerdings nur eine begrenzte Anzahl an Seitenzuordnungen. Sind diese aufgebraucht, so muss beim nächsten Zugriff ein Eintrag entfernt werden, bevor ein neuer hinzugenommen werden kann. Jeder Prozess besitzt seine eigene Seitentabelle. Auf x86-basierten Rechnern wird im TLB allerdings nicht gespeichert, welchem Prozess ein bestimmter Eintrag gehört (keine *Adressraum-Tags*). Um falsche Zuordnungen zu vermeiden, muss also der TLB nach jedem Adressraumwechsel gelöscht werden.

Diskussion

Für immer wieder verwendete Standardfunktionen, bspw. das Einlesen von Dateien, die Manipulation von Zeichenketten oder aber auch die Verwaltung von Speicher, nutzen Softwareentwickler typischerweise Bibliotheken. In Linux-Systemen wird häufig die *GNU C Library* (*glibc*) als Standardbibliothek verwendet. Deren Speicherverwaltung nutzt zur Allokation den Algorithmus von Douglas Lea [42]. Dieser bietet zwar eine sehr gute mittlere Performanz, ist aber wegen seiner $\mathcal{O}(n)$-Komplexität nicht für Echtzeitsysteme geeignet (vgl. [43], S. 1). Falls in herkömmlichen Linux-Systemen innerhalb von Echtzeittasks dynamisch Speicher angefordert werden soll, muss folglich auf einen anderen Speicherallokator zurückgegriffen werden (vgl. [4]). Wurde der Speicher allerdings vor Eintritt in die Echtzeitphase angefordert, so ist dessen Nutzung auch innerhalb des Echtzeittasks möglich. Dabei muss allerdings darauf geachtet werden, dem Betriebssystem die Auslagerung des Speichers zu verbieten. Die Aufrufe `mlock()` bzw. `mlockall()` bieten in Linux-Systemen diese Funktionalität[4].

Wie zu Beginn bereits erwähnt, sind in Echtzeitsystemen Zeitverzögerungen jeglicher Art kritisch. Moderne Hardware erzeugt diese bspw. durch den Einsatz von Caches für Prozessor und MMU. Die entstehenden Verzögerungen setzen sich dabei aus direkten und indirekten Anteilen zusammen. Indirekte Kosten durch Adressraum- oder Prioritätslevelwechsel tragen allerdings besonders zur Gesamtbilanz bei. Während das Löschen bzw. Invalidieren des TLB sehr schnell durchgeführt werden kann, ist das Wiederbefüllen desselbigen sehr aufwendig und nimmt einige Prozessor- sowie Speicherzyklen in Anspruch. Dieser Vorgang ist auf x86-basierter Hardware bei jedem Adressraumwechsel, genauer beim Setzen der neuen Seitentabelle, nötig. Dies geschieht vollkommen transparent mit Hardwareunterstützung. Der Anwender nimmt letztendlich nur die verzögerte Ausführung der Instruktionen wahr. Entwickler von Echtzeitsystemen müssen diese Tatsache daher beim Entwurf berücksichtigen. Da in monolithischen Systemen jedoch die gesamten Betriebssystemfunktionen ohne Adressraumwechsel in Anspruch genommen werden können, sind keine besonders hohen Kosten aufgrund fehlender TLB-Einträge zu erwarten. Selbstverständlich treten diese zwar auch in Linux auf, werden in ihrer Häufigkeit hier aber hauptsächlich durch den Umfang (*working set*) der ausgeführten Coderoutinen bestimmt. Selbiges gilt auch für Kosten durch den Einsatz von CPU-Caches.

[4] weiterführende Informationen in [30], S. 758ff

Nahezu die gesamte Funktionalität der Speicherverwaltung in DROPS ist in Userspace-Servern außerhalb des Mikrokerns implementiert. Die dabei verwendete Architektur ist sehr flexibel und erlaubt den Aufbau einer hierarischen Struktur. Beim Systemstart wird der physische Speicher einer zentralen Komponente, *Sigma0*, auch *Root-Pager* genannt, zugeschrieben. Ein Pager übernimmt grundsätzlich die Verwaltung von Speicher. Dieser kann aber bei Bedarf Teile davon an weitere Pager übertragen. Diese wiederum können den ihnen zugeteilten Speicher an weitere Pager verteilen. Die so gewonnene Pager-Hierarchie erlaubt eine feingranulare Aufteilung des Speichers. Jeder Bereich kann dabei eigene Politiken, beispielsweise zur Auslagerung von Speicherseiten, implementieren und somit auf konkrete Anforderungen eines Anwendungsbereichs eingehen. Abbildung 4.3 illustriert ein mögliches Anwendungsbeispiel: L^{4}Linux implementiert hierbei die Standard-Linux-Paging-Politik mit zusätzlicher Swap-Partition, währenddessen ein weiterer Pager eine Echtzeitstrategie umsetzt. Jedem Thread wird in DROPS genau ein Pager zugeordnet (ein Pager kann aber für mehrere Threads zuständig sein). Der Pager wiederum ist auch ein Thread. Tritt ein Seitenfehler auf, so wird zunächst der kerninterne *Pagefault-Handler* aufgerufen. Dieser wiederum informiert den Pager des fehlgeschlagenen Threads per IPC über dieses Ereignis. So erhält dieser Informationen darüber, an welcher Adresse der Seitenfehler aufgetreten ist und welche Codestelle dabei gerade ausgeführt wurde. Nun werden die angeforderten Seiten beschafft und anschließend in den Adressraum des fehlgeschlagenen Threads eingebunden. Anschließend kann dieser die letzte Instruktion wiederholen und somit mit der Bearbeitung fortfahren.

Um in L4-Applikationen zusammenhängenden und vor dessen Auslagerung geschützten Speicher anzufordern, kann der Nutzer unterschiedliche Dataspace-Manager benutzen.

Als C-Bibliothek kommt standardmäßig *uClibc* zum Einsatz. Deren `malloc()`-Implementierung nutzt in der Grundeinstellung ebenfalls Lea's Algorithmus, der bei Bedarf aber ausgetauscht werden kann.

In mikrokernbasierten Architekturen werden große Teile des Betriebssystems in unterschiedlichen Prozessen implementiert. Dadurch findet prinzipbedingt eine umfangreiche Interprozesskommunikation statt. Diese wiederum impliziert eine Vielzahl von Adressraumwechseln. Die tatsächlichen Kosten werden zum Einen von der verwendeten Hardware bestimmt. Des Weiteren unterscheiden sie sich von Applikation

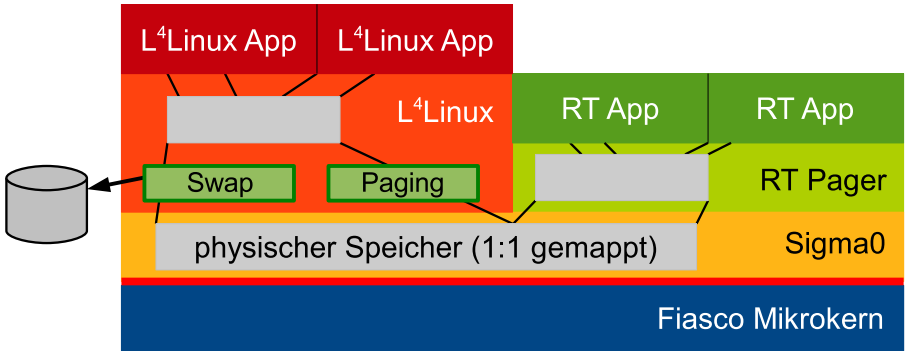


Abb. 4.3: Flexibles, hierarisches Paging in DROPS (vgl. [44], Folie 28f)

zu Applikation und hängen direkt von der Frequenz ab, mit der die Kommunikation (und damit auch die Adressraumwechsel) stattfinden (vgl. [38], S. 9). Dabei unterscheiden sich unterschiedliche Plattformen zum Teil erheblich bei der Cache- bzw. TLB-Verwaltung. Während 32-Bit x86-Hardware bspw. physisch indizierten Cache nutzt, wird dieser bei ARM-basierten Prozessoren bis zur Version 5 über virtuelle Adressen angesprochen. Eine für Adressraumwechsel gewissermaßen optimierte Architektur würde physisch indizierten Cache nutzen sowie eine Möglichkeit zur softwarebasierten Manipulation der Einträge bieten. Auch die TLB-Einträge ließen sich per Software verwalten und würden Adressraum-Tags unterstützen, sodass diese bei einem Adressraumwechsel nicht vollständig geleert werden müssten. Letzteres kann von 32-Bit x86-Hardware nicht zur Verfügung gestellt werden. Hohmuth et. al beschreibt in [41] jedoch zwei Techniken, die auch auf solcher Hardware die exklusive Zuweisung von Cache- und TLB-Einträgen ermöglichen. Zur Zeit werden diese jedoch nicht implementiert.

Fazit

Die Verwaltung der Ressource Speicher ist eine zentrale Aufgabe von Betriebssystemen. Dabei kann sich die konkrete Implementierung in verschiedenen Systemen stark voneinander unterscheiden. Dies kann allerdings nicht die starke Abhängigkeit von der verwendeten Hardware verbergen. Zusammenfassend sind somit nur geringe architekturbedingte Auswirkungen auf die Echtzeitfähigkeit zu erwarten. Erhöhte Flexibilität beim Speichermanagement in DROPS bezahlt der Nutzer mit zusätzlichem Kommunikationsaufwand beispielsweise bei der Fehlerverarbeitung. Da

Seitenfehler in Echtzeitanwendungen jedoch ohnehin unerwünscht sind, ist dies hier nur von sekundärer Relevanz. Trotz aufwendiger Analysen und komplexer Hardwaremodelle ist der Umfang, der durch fehlende TLB-Einträge verursachten Kosten, schwer einzuschätzen. Zumindest auf x86-basierter Hardware, die keine Zuordnung von TLB-Einträgen zu Adressräumen zulässt, sind diese in Mikrokernarchitekturen höher einzustufen.

4.1.5 Zeitbasis

Dieser Abschnitt befasst sich mit der Frage, wie präzise in einem Echtzeitcomputersystem beispielsweise Zeiten gemessen oder periodische Aufgaben angestoßen werden können. Diese Thematik steht in engem Zusammenhang mit dem in Teil 4.1.1 besprochenen Scheduling.

Grundlagen

In Echtzeitsystemen spielt Zeit eine entscheidende Rolle. Oft muss diese erfasst, miteinander verglichen oder ausgewertet werden. Kritische Applikationen werden periodisch ausgeführt, dabei wird überwacht, dass diese nicht zuvor festgelegte Fristen überschreiten. Für die Koordination der gesamten Funktionalität ist das Betriebssystem zuständig. Dabei nimmt der *System-Timer* eine zentrale Rolle ein. Er dient als Zeitbasis. Werden Wartezeiten beispielsweise in Millisekunden angegeben, so ist deren tatsächliche Auflösung entscheidend von der Granularität dieses Timers abhängig. Angenommen sie beträgt $976\mu s$[5] und ein Anwender programmiert einen Timeout von $1ms$, so dauert dieser tatsächlich rund $2ms$.

Wie bereits angedeutet, hängt die erzielte Auflösung maßgeblich von der zur Verfügung stehenden Timerhardware ab. In x86-basierter Hardware kommt dafür entweder der PIT (*programmable interrupt timer*), der Timerbaustein des APIC (*advanced programmable interrupt controller*) oder die Echtzeituhr (RTC) zum Einsatz. Neuere Systeme bieten weiter Unterstützung für den sogenannten *HPET* (*high precision event timer*).

[5] *RTC* (*real time clock*) auf x86-Hardware als Zeitbasis

Bei der Taktgenerierung unterscheidet man zwischen zwei Betriebsmodi: periodischer Modus und Einzelbetrieb. Im periodischen Modus wird die Timerhardware einmalig programmiert und erzeugt dann fortlaufend sogenannte Timer-Interrupts. Im Einzelbetrieb (*one shot mode*) hingegen wird die Hardware nach jedem Auslösen erneut auf das nächste Ereignis eingestellt. Vorteil dieser Methode ist die hohe Flexibilität. Außerdem werden im Vergleich zum periodischen Modus nicht unnötig viele Interrupts erzeugt. Nachteilig ist allerdings die erhöhte Komplexität der Routinen zur Timerprogrammierung. Des Weiteren ist diese unter Umständen sehr aufwendig, beispielweise wenn die dazugehörigen I/O-Register außerhalb der CPU liegen. Dies hat zur Folge, dass sich die Laufzeit des Timer-Interrupt-Handlers erheblich erhöht. Zu kurze Timerintervalle können zusätzlich zu einem unbenutzbaren System führen. Dies ist der Fall, wenn die Abarbeitung der Timer überproportional viel CPU-Zeit in Anspruch nimmt.

Zur Zeitmessung kommt auf x86-basierter Hardware fast ausschließlich der, ab der Pentium-Familie eingeführte, *TSC* (*time stamp counter*) zum Einsatz. Dies ist ein 64-Bit Register, welches im Systemtakt inkrementiert wird. Es eignet sich dadurch für sehr präzise Messungen im Nanosekundenbereich. Außerdem ist der Zugriff auf dieses Register auch aus dem Userspace möglich.

Diskussion

Die ab Version 2.6.24 im Linux-Kern zur Verfügung stehenden *hrtimer* (*high resolution timer*) unterstützen feingranular aufgelöste Ereignisse. Auf x86-basierter Hardware nutzen diese den lokalen APIC-Controller als Zeitbasis. Dieser wird dabei im One-Shot-Mode konfiguriert. Der Einsatz der *hrtimer* entkoppelt normale Systemtimer vom periodischen Schedulertick. Für Systeme, die auf Ereignisse mit einer Auflösung $\leq 1ms$ reagieren sollen, ist deren Verwendung in jedem Fall zu empfehlen. Die simple Erhöhung der Schedulertickfrequenz würde zwar prinzipiell auch funktionieren, hätte aber automatisch einen erhöhten Overhead durch vermehrte Scheduleraufrufe zur Folge.

Als Schedulingfrequenz haben sich $1000Hz$ etabliert. Zeitscheiben werden beim Round-Robin-Scheduling daher mit einer Granularität von $1ms$ verwaltet.

Für zeitunkritische Anwendungen kommt in DROPS bzw. Fiasco in der Grundeinstellung der PIT-Timer als Taktgeber zum Einsatz. Dieser sollte nur für den periodischen Betrieb programmiert werden, da die Kosten zur Umprogrammierung verhältnismäßig hoch sind. Etwas besser verhält sich der APIC-Timer in dieser Disziplin: Seine Programmierung verlängert den Codepfad nicht unverhältnismäßig. Er eignet sich somit für den Einzelbetrieb und ermöglicht beispielsweise auch Wartezeiten $\leq 1ms$. In der Grundeinstellung wird der Scheduler ebenfalls fest mit einer Frequenz von $1000Hz$ aufgerufen. Eine Heuristik zur Erkennung von Überlastsituationen, durch zu viele Timerevents, ist nicht implementiert.

Fazit

Der Systemtakt in Echtzeitsystemen ist vergleichbar mit dem Herzschlag bei Lebewesen. Dieser darf sich weder verzögern noch ganz wegbleiben. Aufgrund der starken Hardwareabhängigkeit unterscheidet sich die Umsetzung in beiden betrachteten Systemen kaum voneinander.

4.1.6 Mess- und Bewertungsverfahren

Um ein gegebenes Computersystem hinsichtlich dessen zeitlicher Eigenschaften bewerten zu können, muss dieses einer *WCET-Abschätzung* (*worst case execution time evaluation*) unterzogen werden. Die WCET für einen Task ist die obere Grenze für die Zeit zwischen Taskstart und Taskende. Dabei müssen alle möglichen Eingaben und Ausführungsszenarien berücksichtigt werden. Die Frist für einen gegebenen Task kann nur garantiert werden, wenn die WCET für alle (in der Interaktion) involvierten Programmteile und Subsysteme bekannt ist. Zusätzlich zum Wissen über Anwendungsprogramme ist somit auch die Abschätzung des administrativen Overheads eines Betriebssystems (WCAO) vonnöten (vgl. Abschnitt 2.1: Evaluation).

In der Literatur werden zwei grundsätzliche Ansätze zur Bestimmung der maximalen Ausführungszeit einer bestimmten Coderoutine unterschieden: statische oder dynamische Programmanalyse (vgl. [2], S. 86). Erstere versucht dieses Ziel durch Offline-Analyse zu erreichen, also ohne die entsprechenden Instruktionen jemals tatsächlich auszuführen. Diese Methode verspricht sehr genaue Ergebnisse, setzt aber

ein präzises Modell der verwendeten Hardware voraus. Dieses sollte möglichst alle Systemkomponenten, wie Prozessor, Cache, TLB, Bussysteme sowie Ein- Ausgabesysteme, abbilden. Des Weiteren ist zu bemerken, dass das Problem, ob ein beliebiges sequenzielles Programm terminiert, im Allgemeinen unentscheidbar ist (Halteproblem für Turingmaschinen). Betrachtet man beispielsweise folgendes Programmstück, so ist es nicht möglich festzustellen, nach wie vielen Durchläufen bzw. ob überhaupt, der Ausdruck *expression* unwahr wird.

Algorithmus 2 Terminierung einer while-Schleife

```
1: while expression do
2:     something
3: end while
```

Aus diesem Grund muss der Quelltext bestimmten Anforderungen genügen, um überhaupt statisch analysiert werden zu können (vgl. [45], S. 2, [2], S. 87):

1. keine Schleifen mit unbegrenzter Abbruchbedingung,
2. keine rekursiven Funktionsaufrufe und
3. es dürfen keine dynamischen Datenstrukturen benutzt werden.

Die dynamische Programmanalyse basiert grundsätzlich auf der Idee, den zu untersuchenden Code mit unterschiedlichen Eingaben auszuführen und damit das Muster zu identifizieren, welches zur längsten Abarbeitungszeit führt. Dabei ist es entscheidend, während der Analyse für realistische Umstände, also eine möglichst große Code- und Datenabdeckung, zu sorgen (z. B. hohe Systemlast erzeugen, Speicher-, Cache- und TLB-Zugriffe provozieren). Nachteilig an der dynamischen Methode ist deren Unsicherheit. Es gibt keine Garantie dafür, dass auch tatsächlich der schlechteste Fall abgedeckt wurde.

In der Praxis wird heute fast ausschließlich auf dynamische Analyse gesetzt. Diese soll daher auch in der vorliegenden Arbeit Verwendung finden. Da kein detailliertes Hardwaremodell benötigt wird, sind die entwickelten Methoden leichter auf andere Hardwareplattformen übertragbar. Konkrete Verfahren, deren Umsetzung sowie die verwendete Hardware werden in Kapitel 6 vorgestellt und umgesetzt.

4.1.7 Fazit

In diesem Kapitel wurden Betriebssystemkomponenten mit direktem Einfluss auf Echtzeiteigenschaften untersucht. Aufgrund hoher Hardwareabhängigkeit sind nur marginale Unterschiede in der Implementierung der Kernkomponenten in beiden Architekturen erkennbar. In mikrokernbasierten Systemen ist dennoch ein nicht zu unterschätzender Overhead durch die Einführung getrennter Adressräume festzustellen. Die genaue Höhe hängt zum Einen von der verwendeten Hardware (bspw. Cache- und TLB-Architektur), zum Anderen von der konkreten Applikation bzw. deren Kommunikationsmuster ab. Der erhöhte Aufwand führt jedoch zu längeren Laufzeiten, da in jedem Fall mehr Code ausgeführt wird.

Die Verlagerung von Betriebssystemfunktionalität in den Userspace verringert die Komplexität des Kerns erheblich. Das Gesamtsystem wird flexibler und die Vorhersagbarkeit erhöht sich, da Gerätetreiber oder andere Kernmodule beispielsweise nicht einfach sämtliche Interrupts sperren können.

Im Gegensatz zu universellen Betriebssystemen wird in Echtzeitrechnern oft auf dynamische Methoden zur Erstellung von Tasks und zur Allozierung von Speicher verzichtet. Sollen diese zum Einsatz kommen, so muss bei den dafür verantwortlichen Komponenten auf deterministisches Verhalten geachtet werden.

4.2 Verlässlichkeitsanalyse

Nachdem Echtzeitaspekte in monolithischen- bzw. mikrokernbasierten Betriebssystemen diskutiert wurden, konzentriert sich dieses Kapitel nun auf die Eignung selbiger für den Einsatz in verlässlichen Computersystemen. Hierfür sollen zunächst Gründe für den Ausfall von Rechensystemen erörtert und klassifiziert werden. Im Anschluss daran erfolgt schließlich die Analyse beider Architekturprinzipien. Hierbei wird untersucht, inwieweit Hilfsmittel zur Erhöhung der Systemverlässlichkeit, genauer zur Fehlerbehandlung und Fehlervermeidung, unterstützt werden.

Motivation

In modernen industriellen Anlagen werden Computersysteme zur Ablaufsteuerung oder Prozessüberwachung eingesetzt. Diese sind dabei häufig als *Single-Master-Systeme* konzipiert. Ein Ausfall des Masters stellt ein besonders hohes Risiko dar, da dies mit einem Gesamtausfall der Anlage gleichzusetzen wäre (*single point of failure* oder *SPOF*).

Ziel verlässlicher Systeme ist es, auch unter Einfluss von Fehlern, einen Service (bis zu einem gewissen Level) bereitstellen zu können. Für diesen Zweck kommen während der Entwicklungsphase unter anderem Techniken zur Fehlervermeidung zum Einsatz.

Darüber hinaus werden Fehler auch im laufenden Betrieb behandelt. Diese müssen hierfür zunächst erkannt werden. Anschließend werden Fehler typabhängig behoben oder zumindest maskiert. Somit soll ein Wirken über die Systemgrenzen hinweg ausgeschlossen werden (vgl. Erkennung und Behebung von Fehlern in Kapitel 2.2).

Die Ursachen der Ausfälle sind dabei recht vielseitig. Fehlerhafte Systemsoftware ist jedoch für einen Großteil aller Defekte in Computersystemen verantwortlich. In modernen Betriebssystemen bestehen ca. zwei Drittel der gesamten Codebasis aus Gerätetreibern.

Studien haben gezeigt, dass diese zusätzlich einer besonders hohen Fehlerrate (drei bis sieben mal höher als anderer Code) unterliegen (vgl. [46]). Durch Integration der oft durch Drittanbieter gelieferten Treiber in den Systemkern, führen Fehler hier leicht zum Ausfall des Computersystems. Grund dafür ist häufig die Tatsache, dass

Gerätetreiber oder andere systemnahe Komponenten mit zu hohen Rechten ausgestattet sind. Dabei sind diese zur Erfüllung der Funktionalität nicht nötig und steigern damit (bei unsachgemäßer Verwendung) die Ausfallgefahr. Die Bereitstellung von geeigneten Isolationsmechanismen für Gerätetreiber und andere Systemkomponenten ist für verlässliche Computersysteme daher essenziell.

Im Folgenden soll nun am Beispiel von Treibern erörtert werden, welche Ursachen für fehlerhaftes Verhalten verantwortlich sind und wie diesen in unterschiedlichen Betriebssystemarchitekturen begegnet werden kann.

4.2.1 Einflussfaktoren

Gerätetreiber sind das Bindeglied zwischen Hardware, Betriebssystem und Anwendungssoftware. Obwohl Treiber auf sehr niedrigem Abstraktionsgrad operieren, ist es dennoch nicht notwendig, uneingeschränkt Zugriff auf sämtliche Systemressourcen zu gewähren.

In [47] klassifiziert Herder et al. für Gerätetreiber typische Operationen bzw. Arbeitsabläufe. Wie hierin beschrieben wird, bringt die uneingeschränkte Nutzung jedoch Gefahren mit sich. Ein böswilliger Anwender oder ein fehlerhaft arbeitendes Programm könnte dies ausnutzen, um die korrekte Funktionsweise des Computersystems zu stören. Im Folgenden werden die erwähnten Aspekte kurz aufgelistet und anschließend in Kapitel 4.2.2 näher analysiert.

Prozessornutzung Es ist ratsam, die Nutzung der CPU durch Gerätetreiber zu kontrollieren. So sollte zum Einen die CPU-Zeit begrenzt, und zum Anderen darauf geachtet werden, dass Treiber nicht uneingeschränkt privilegierte CPU-Instruktionen ausführen können.

Speicherzugriff Gerätetreiber tauschen häufig Daten mit anderen Systemkomponenten (bspw. mit Kommunikationsprotokollen) aus. Diese sollten dabei nur Zugriff auf ausgewählte Speicherbereiche haben. Es ist zudem empfehlenswert, deren DMA-Aktivitäten zu überwachen, da hierdurch ebenfalls der Zugriff auf den gesamten Speicher möglich ist.

Geräte Ein-/ Ausgabe Zugriffe auf I/O-Ports, Register, Gerätespeicher sowie die Manipulation der Interruptlogik sind potenziell gefährliche Operationen, da

durch sie bspw. alle Interrupts ausgeschalten werden können.

Systemservices Treiber kommunizieren zum Einen mit ihren Nutzern, zum Teil aber auch mit anderen Systemkomponenten. Die IPC mit fehlerhaften oder gar böswilligen Partnern stellt somit eine besondere Herausforderung für die Systemsicherheit dar.

Nach Herder lässt sich durch eine Einschränkung der Rechte von Gerätetreibern die Verlässlichkeit von Computersystemen erheblich steigern. Nachfolgend soll daher untersucht werden, inwieweit monolithische bzw. mikrokernbasierte Betriebssysteme die Isolation von Treibern unterstützen.

4.2.2 Fehlerbehandlung

Unter Fehlerbehandlung versteht man in der Softwaretechnik allgemein aktive Techniken zur Unterstützung der Systemverlässlichkeit. Dabei werden verschiedene Methoden kombiniert, die das gemeinsame Ziel verfolgen, Fehler zu entfernen oder zu maskieren.

Die Analyse dieser Techniken findet nun auf Grundlage der in Kapitel 4.2.1 erarbeiteten Einflussfaktoren statt. Die von Treibern ausgehenden Gefahren werden zu diesem Zweck noch einmal kurz skizziert. Im Anschluss daran werden Auswirkungen in beiden betrachteten Architekturprinzipien diskutiert.

Prozessornutzung

Gerätetreiber laufen in monolithischen Systemen typischerweise im Kernmodus[6]. Die Ausführung von CPU-Instruktionen zur Manipulation systemkritischer Komponenten ist somit uneingeschränkt möglich. Fehlerhafte Treibersoftware kann somit zum Beispiel durch Umprogrammieren der Timerhardware oder durch Ausführung der `cli`-Anweisung (Sperrung der Interrupts) die Systemtaktgenerierung stoppen.

Die Zuordnung von Prozessorzeit erfolgt in Linux-Systemen entweder durch einen Benutzerprozess (z. B. nach einem Systemaufruf), den Aufruf eines Kernelthreads oder das Eintreffen asynchroner Ereignisse (Interrupts). Interrupt-Service-Routinen

[6] Ausnahmen sind in Linux bspw. einige Dateisystemtreiber (*FUSE, filesystem in userspace*)

laufen zunächst im privilegierten Interrupt-Kontext. Sie sind somit, ähnlich wir kritische Abschnitte, potenzielle Angriffspunkte, um den Prozessor für längere Zeit zu blockieren.

In mikrokernbasierten Architekturen werden Treiber im Benutzermodus implementiert und unterscheiden sich dadurch nicht von gewöhnlichen Applikationen. Der Aufruf privilegierter Instruktionen ist nur dem Mikrokern vorbehalten oder wird von diesem zumindest überwacht. Falls nötig, können Treiber durch Systemaufrufe über definierte Schnittstellen diese Funktionen nutzen. Die Zuordnung von Prozessorzeit geschieht über den Betriebssystemscheduler. Bei korrekter Vergabe der Prioritäten und Konfiguration der Zeitscheiben kann ein Blockieren der CPU durch Treibersoftware somit ausgeschlossen werden.

Speicherzugriff

Der Zugriff auf Speicher durch Treiber kann in Systemen wie Linux nicht eingegrenzt werden. Diese sind somit zum Beispiel in der Lage, wichtige Datenstrukturen zu überschreiben oder beliebige Funktionen aufzurufen. Grundsätzlich ist es zwar möglich, softwarebasiert alle Lade- und Speicheroperationen (oder auch andere CPU-Instruktionen) zu überwachen und dadurch nur Zugriff auf bestimmte Bereiche zu gewähren. Der Einsatz solcher Techniken erfordert allerdings einen vergleichsweise hohen Aufwand zur Übersetzungs- und Laufzeit.

Geräte nutzen zur Entlastung des Hauptprozessors für große Datentransfers häufig DMA. Treiber bzw. Gerät haben dabei vollständigen Zugriff auf den physischen Speicher des Rechners. Abhilfe für dieses Problem bringen bspw. sogenannte *IOMMUs* (*input/output memory management unit*). IOMMUs übersetzen für das Gerät sichtbare Adressen in physische Adressen. Ähnlich wie MMUs gewähren sie so Schutz vor ungewollten Speicherzugriffen. In monolithisch konzipierten Betriebssystemen garantiert dieser Mechanismus jedoch nur Schutz vor fehlerhafter Hardware. Da Gerätetreiber selbst im Kernmodus laufen, können diese die IOMMU auch direkt umprogrammieren und dadurch deren Schutzmechanismen umgehen.

In mikrokernbasierten Architekturen kommen die Vorteile hardwareunterstützter Isolation durch die MMU vollkommen zum Tragen. Letztere übersetzt Zugriffe auf

virtuelle Adressen im eigenen Adressraum auf deren physische Pendants. Speicher anderer Prozesse (also auch anderer Treiber) kann so nicht manipuliert werden.

Die Vorteile einer IOMMU werden ebenfalls vollständig unterstützt. Gerätetreiber, die DMA nutzen möchten, programmieren die IOMMU unter Zuhilfenahme einer vertrauenswürdigen Instanz. Der Zugriff auf Speicher lässt sich somit bspw. nur auf den treibereigenen Adressraum begrenzen.

Geräte Ein-/ Ausgabe

Die Ausführung der Gerätetreiber im privilegierten Prozessormodus ermöglicht diesen in monolithischen Systemen uneingeschränkten Zugriff auf I/O-Ports, Register oder Gerätespeicher. Treiber können somit Hardware beliebig umprogrammieren, auf Bussysteme zugreifen oder z. B. die parallele Schnittstelle benutzen.

Darüber hinaus ist selbst die Manipulation von Interrupthardware möglich.

Beim Zugriff auf Ein/-Ausgabehardware erhöht der Einsatz von mikrokernbasierten Betriebssystemen die Verlässlichkeit der Gesamtarchitektur. Lese- und Schreiboperationen auf diese müssen explizit vom Kern gestattet werden, z. B. durch Setzen der entsprechenden Berechtigungen im *Task-State-Segment*[7].

Obwohl die Behandlung der Top-Half Interrupt-Handler im Kern durchgeführt wird, findet die gesamte gerätespezifische Abarbeitung im Userspace statt. Die Aufteilung in Top- und Bottom-Half Handler ist zwar auch in monolithischen Systemen üblich, findet dort aber üblicherweise aus Gründen der Performanz statt[8].

Systemservices

Gerätetreiber im Kernmodus können durch direkte Funktionsaufrufe beliebige Systemkomponenten nutzen. Dabei besteht die Gefahr, dass sich Fehler auf andere Komponenten übertragen und somit im System ausbreiten können.

Laufen Gerätetreiber in getrennten Adressräumen, so können diese keine Funktionen über Adressraumgrenzen hinweg aufrufen. Subsysteme sind somit voneinander isoliert

[7] bei x86-basierter Hardware
[8] Bottom-Half Interrupt-Handler laufen ebenfalls im Kernmodus

und die Ausbreitung von Fehlern wird damit zunächst auf die eigene Komponente beschränkt. Durch Fehler im Betriebssystemkern selbst oder durch Kommunikation mit anderen Prozessen können sich jedoch auch hier Fehler fortpflanzen. Durch gewissenhafte Konfiguration lässt sich der Einfluss von IPC jedoch gut begrenzen. Kommunikationsmechanismen sind in L4-basierten Systemen sehr flexibel parametrierbar und unterstützten bspw. Timeouts, um fehlerhafte Kommunikationspartner identifizieren zu können. Zusätzlich können Sender bzw. Empfänger von IPC-Nachrichten explizit festgelegt werden. Dies ermöglicht auch die Realisierung von definierten Kommunikationspolitiken (bspw. Komponente A darf nicht mit Komponente B, dafür aber mit Komponente C kommunizieren).

Fazit

Gerätetreiber sind nicht selten für Ausfälle in Computersystemen verantwortlich. Auslöser dafür ist der Umstand, dass diese häufig mit zu hohen Systemrechten ausgestattet sind. Aus diesem Grund fand im vorliegenden Kapitel eine Untersuchung typischer Arbeitsabläufe von Treibern statt. Dabei wurden mögliche Folgen sowie Mechanismen zur Fehlerbehandlung in zwei Betriebssystemarchitekturen diskutiert. Es konnte gezeigt werden, dass mikrokernbasierte Systeme, durch Abschottung von Treibern in einzelne Adressräume, die Systemverlässlichkeit erhöhen. Einige Mechanismen lassen sich überhaupt erst effizient durch die Implementierung von Gerätetreibern im Userspace umsetzen.

Ein weiterer Aspekt zur Fehlerbehandlung ist das Einspielen von Softwareupdates. Wurde ein Programmfehler entdeckt und ein passender *Patch* entwickelt, so muss dieser noch auf das System angewendet werden. In mikrokernbasierten Architekturen können Server unabhängig voneinander angehalten und wieder neu gestartet werden. Der Austausch von Gerätetreibern ist so selbst im laufenden Betrieb möglich. Es existieren zwar auch Technologien zum Online-Update von Linux-Systemen. *Ksplice* [48] zum Beispiel tauscht Objektcode während der Laufzeit aus. Verändern Softwareupdates jedoch die Semantik des Programms, so kann diese Methode nicht angewendet werden.

4.2.3 Fehlervermeidung

Fehlervermeidung meint im Kontext der Softwareentwicklung die Kombination verschiedener Techniken mit dem Ziel, die Wahrscheinlichkeit für das Auftreten von Defekten zu minimieren. Die dabei verwendeten Methoden werden häufig schon während der Entwicklung des Systems appliziert. Zur Bewertung werden Softwarequalitätsfaktoren betrachtet. Dies sind nicht-funktionale Anforderungen an die Software, welche inhärent zur Verbesserung der Systemverlässlichkeit beitragen.

Fehlerprävention

Fehlerprävention umfasst Techniken zur Qualitätsverbesserung von Soft- und Hardware während der Entwicklung und Herstellung. Im Bereich Software wird häufig die Verwendung von sogenannten Vorgehensmodellen gefordert. Dabei soll sichergestellt werden, dass über einen „qualitativ hochwertigen Prozess der Produkterstellung die Entstehung von qualitativ hochwertigen Produkten begünstigt" wird [49]. Hierzu zählen bspw. strukturiertes Programmieren, aber auch Methoden des objektorientierten Paradigmas. Beispiele hierfür sind Konzepte wie *Information Hiding*, Kapselung, Modularisierung sowie Kopplung und Kohäsion [50].

Durch Verlagerung von Systemservices in den Userspace wird die Komplexität des Betriebssystemkerns in mikrokernbasierten Architekturen erheblich verringert. Klare, einheitliche Kernschnittstellen abstrahieren von plattformspezifischen Interfaces und begünstigen somit eine modulare Systemstruktur. Die Gesamtarchitektur wird flexibler und kann dadurch für den Einsatz in unterschiedlichen Anwendungsszenarien (z. B. in eingebetteten Systemen) konfiguriert werden.

Gerade in monolithischen Systemen ist eine klare, modulare Struktur beim Entwurf von Software unerlässlich. Würde dies nicht geschehen, wären komplexe Projekte wie bspw. Linux nicht denkbar. Sie würden an der Umsetzung oder spätestens bei der Wartung scheitern. Die erfolgreiche Realisierung dieses Grundkonzepts in Linux verdeutlicht allein die Anzahl an Hardwareplattformen, für die dieses Betriebssystem verfügbar ist. Dennoch werden immer wieder Anwendungsschnittstellen (API) zu Kernkomponenten verändert. Dies zeugt zwar von einer hohen Dynamik in der

Entwicklung, generiert bei der Portierung von Treibern aber auch immer wieder neue Fehlerquellen.

Fehlerentfernung

Die in der Entwicklungsphase von Softwarekomponenten am häufigsten durchgeführten Tätigkeiten zur Fehlerentfernung sind Softwaretests. Dabei werden empirisch Informationen darüber gesammelt, inwieweit das entwickelte Produkt den Ansprüchen bzw. Anforderungen, die bei dessen Entwurf aufgestellt wurden, entspricht. Über mehrere Iterationen hinweg wird hierdurch die Systemqualität verbessert. Tests sind daher unerlässliche Instrumente zur Verbesserung der Verlässlichkeit. Der Aufwand dieser Untersuchungen ist dabei allerdings nicht unerheblich. Er nimmt einen Großteil der veranschlagten Zeit in Anspruch und hängt entscheidend von der Komplexität der betrachteten Komponenten sowie deren Schnittstellen ab.

Eine weitere Möglichkeit zur Qualitätssicherung in Softwaresystemen sind Validierung und Verifikation. Letztere beschreibt dabei den Prozess zur Überprüfung, ob die entwickelte Software (oder auch nur Teile davon) den zuvor spezifizierten Anforderungen genügt (*"Wird die Software richtig entwickelt?"*). Validierung meint hingegen die Überprüfung der Software hinsichtlich deren Eignung für den (gewünschten) Einsatzzweck (*"Wird die richtige Software entwickelt?"*).

Softwaretests, wie Unit-Tests, Regressions-Tests oder Usability-Tests können sowohl in monolithischen als auch in mikrokernbasierten Architekturen durchgeführt werden. Sie steigern die Softwarequalität jedoch nur, wenn sie auch korrekt ausgeführt werden und eine hohe Abdeckung erreichen. Durch enge Kopplung der Komponenten in monolithischen Systemen erhöht sich die Anzahl der logischen Zustände, die das Programm einnehmen kann, erheblich. Ein ausreichend hoher Abdeckungsgrad kann aus diesem Grund nur schwer erzielt werden.

Die Codebasis des Kernels ist in mikrokernbasierten Systemen typischerweise erheblich schlanker als in deren monolithischen Pendants. Zum Vergleich: statt mit vier Millionen Zeilen Kernelcode in Linux Version 2.6.17 [51] kommt Minix3 [17] mit knapp 4.000 Zeilen aus.

Der L4-basierte Mikrokern *seL4* besteht aus nur 8.700 Zeilen C und 600 Zeilen

Assembler Code. Dessen Entwicklern ist es gelungen, die C-Implementierung als ersten Betriebssystemkern überhaupt formal zu verifizieren [52]. Eine Verifizierung von monolithischen Kernen ist aufgrund ihrer Ausmaße dagegen kaum vorstellbar.

Fehlervorhersage

Ähnlich wie bei der Entfernung von Fehlern wird auch bei der Fehlervorhersage analysiert, ob die betrachtete Software den spezifizierten Anforderungen entspricht. Dabei wird das Verhalten der Software in Bezug auf das Vorhandensein, die Aktivierung sowie die Konsequenzen von Defekten untersucht. Es wird zum Einen der aktuelle Softwarestand geprüft und mithilfe der durch Systemtests gewonnenen Informationen deren (derzeitige) Verlässlichkeit bewertet (*reliability estimation*). In einer zweiten Phase (*reliability prediction)* wird nun versucht, Aussagen über die zukünftige Verlässlichkeit zu treffen. Hierfür werden unterschiedliche, vom aktuellen Entwicklungsstand abhängige, Techniken eingesetzt. Bevor also bspw. reale Messdaten vorliegen, können u.a. Informationen über den Entwicklungsprozess oder auch Softwaremetriken zur Analyse beitragen (vgl. [53], S. 11).

Kommen bei der Fehlervorhersage bspw. Produktmetriken wie Umfang (z. B. Codezeilen) oder Entwurfsqualität (z. B. Modularität, Kohäsion, Kopplung) zum Einsatz, so sind Mikrokernarchitekturen hierin besser zu bewerten. Aspekte wie Lesbarkeit (Programmierstil), Produktivität bei der Entwicklung oder Entwicklungszeit sind jedoch unabhängig vom zugrunde liegenden Architekturmodell. Insbesondere der letzte Punkt gilt jedoch nur, wenn bei der Analyse auch von einem vergleichbaren Funktionsumfang ausgegangen wird. Dies gilt im Übrigen auch für die gesamte Systemkomplexität. Sie ist in mikrokernbasierten Architekturen nicht automatisch niedriger. Dennoch lässt sich durch modulares Design die Systemkomplexität- und Funktionalität steigern, ohne dadurch gleichzeitig Abstriche bei der Softwarekomplexität- und Qualität hinnehmen zu müssen.

Fazit

Sollen verlässliche Softwaresysteme entstehen, so muss dieses Ziel über alle Entwicklungsphasen hinweg verfolgt werden. Dabei kommt es auf den geeigneten Einsatz

von Vorgehens- oder Entwicklungsmodellen genauso an wie auf hohe Codequalität oder umfangreiche Tests.

Zusammenfassend sind bei allen Aspekten der Fehlervermeidung Vorteile in mikrokernbasierten Architekturen im Vergleich zu monolithischen festzustellen. Die Architektur ist modularer aufgebaut und die APIs wirken stabiler. Die kleinere Codebasis des Betriebssystemkerns, aber auch die Isolation der restlichen Systemkomponenten helfen bei der Durchführung von Softwaretests.

4.2.4 Mess- und Bewertungsverfahren

Soll eine Bewertung von verlässlichen Computersystemen erfolgen, so muss die Effizienz der eingesetzten Hilfsmittel zur Fehlerbehandlung und Fehlervermeidung evaluiert werden. Ähnlich wie bei der Echtzeitanalyse, wird diesbezüglich zwischen statischen und dynamischen Verfahren unterschieden. Erstere beinhalten formale Methoden und Modelle zur Bewertung von Fehlerbehandlungsmechanismen (vgl. [2], S. 248). Bei der Anwendung dynamischer Verfahren wird dagegen angestrebt, durch intensive Tests herauszufinden, ob ein System den Anforderungen genügt und alle Fehler beseitigt wurden. Hierbei werden absichtlich fehlerhafte Eingaben generiert, um somit das Fehlermanagement des Systems zu aktivieren. Diese Methode wird auch als Fehlerinjektion (*fault injection*) bezeichnet.

In der Literatur findet sich eine Vielzahl an Arbeiten, die sich mit der Evaluation der Verlässlichkeit von Betriebssystemen befassen. David et. al prüft in [11] bspw. die Robustheit mehrerer, zum Teil mikrokernbasierter Betriebssysteme. Dabei nutzt er zur Injizierung der Fehler u.a. eine modifizierte Version von *Qemu*[9]. Das in [54] von Arlat et. al entwickelte Tool *MAFALDA* nutzt zur Fehlereinschleusung *Hardwarebreakpoints*. *FAUmachine* [55] ist ein Emulator für Standard-PC-Hardware, der ausschließlich mit dem Ziel entwickelt wurde, diese Systeme mithilfe von Fault-Injection zu bewerten. In [56] werden ebenfalls Betriebssystemservices hinsichtlich ihrer Robustheit bewertet. Diese Arbeiten haben gezeigt, dass Fehler in essenziellen Systemkomponenten (bspw. Gerätetreibern) schwerwiegende Folgen für die Verlässlichkeit haben und leicht zum Systemausfall führen können.

[9] Software zur Emulation von Computerhardware

Verlässliche Systeme müssen demnach aufgetretene Fehler isolieren und geeignete Recoverymechanismen bieten. In zeitkritischen Anwendungen stellt diese Aufgabe eine besondere Herausforderung dar, da neben der Wertedomäne auch zeitliche Aspekte betrachtet werden müssen.

4.2.5 Fazit

In diesem Kapitel wurden monolithische und mikrokernbasierte Betriebssystemarchitekturen am Beispiel von Linux und DROPS hinsichtlich deren Eignung für den Einsatz in verlässlichen Rechensystemen untersucht. Da Steuerungen in industriellen Anlagen oft als Single-Master-Systeme aufgebaut werden, gilt es, diese besonders vor möglichen Ausfällen zu schützen. Hierbei können bspw. mikrokernbasierte Systeme zum Einsatz kommen. Dieses Konzept isoliert Subsysteme voneinander und dämmt dadurch die Ausbreitung von Fehlern ein. Fehlerhaften Gerätetreibern, der Hauptursache vieler Ausfälle, kann dadurch entgegen gewirkt werden. Neben der Unterstützung von Fehlerbehandlungsmechanismen bieten mikrokernbasierte Architekturen auch Vorteile im Bereich der Fehlervermeidung. Kommen diese zum Einsatz, so können Entwickler verlässlicher Systeme die Flexibilität und Qualität ihrer Produkte erhöhen, ohne gleichzeitig deren Komplexität zu steigern.

4.3 Zusammenfassung

Im vorliegenden Kapitel wurden zunächst Echtzeiteigenschaften und später Aspekte der Systemverlässlichkeit in zwei unterschiedlichen Betriebssystemen getrennt voneinander diskutiert. Ausgangsbasis dieser Analyse war die Frage, ob sich die eine oder andere Architektur jeweils besser für einen Einsatzzweck eignet.

Dabei hat sich herausgestellt, dass sich beide Architekturprinzipien grundsätzlich für den Aufbau von Echtzeitsystemen nutzen lassen. Keines der beiden untersuchten Systeme konnte sich in diesem Feld deutlich abheben. Dieses Resultat ändert sich im Bereich der Systemverlässlichkeit jedoch erheblich. Hier haben mikrokernbasierte Systeme aufgrund ihrer Isolationsmechanismen erhebliche Vorteile gegenüber monolithischen Architekturen. Der Aufbau nachweislich verlässlicher Systeme ist unter realistischen Bedingungen sogar nur mit Mikrokernen möglich. Die von der Architektur gebotenen Isolationsmethoden erhält der Anwender allerdings nicht ohne die dadurch implizierten Kosten. Der Kommunikationsaufwand erhöht sich, es wird mehr Code ausgeführt, da das System insgesamt etwas größer wird. Aus diesem Grund kann für Anwendungen mit reinen Echtzeitanforderungen der Einsatz von Mikrokernsystemen nicht uneingeschränkt empfohlen werden. Zeitkritische Applikationen sind jedoch häufig auch sicherheitsrelevant. Die Anforderungen an das Gesamtsystem verändern sich damit schlagartig. Neben der rechtzeitigen Erfüllung von Aufgaben müssen die Systeme zusätzlich falsche Eingaben erkennen und fehlerhafte oder ausgefallene Komponenten maskieren. Selbstverständlich muss bei der Ausführung der Fehlerbehandlungsmechanismen auch auf deren deterministisches Verhalten geachtet werden. Die vorliegende Analyse hat gezeigt, dass monolithisch konzipierte Echtzeitsysteme diesen Anforderungen nur schwer gerecht werden können. Mikrokerne unterstützen jedoch den Spagat zwischen Echtzeit und Verlässlichkeit und ermöglichen somit die Entwicklung von Systemen für unterschiedlichste Anwendungsdomänen.

Kapitel 5

5 Entwurf und Implementierung

Im zweiten Teil der Arbeit sollen nun die im vorherigen Abschnitt durchgeführten theoretischen Betrachtungen experimentell überprüft werden. Hierzu wurde exemplarisch ein für Linux entwickelter EtherCAT-Stack auf ein mikrokernbasiertes System portiert. Diese Arbeit bildet zugleich die Grundlage für den daran anschließenden Vergleich.

Zu Beginn wird in Abschnitt 5.1 die Entwurfs- und Designphase der durchgeführten Portierung beschrieben. Darauf aufbauend erläutert Abschnitt 5.2 die Implementierungsphase.

Der Quellcode sämtlicher erstellter Anwendungen sowie alle Messdaten befinden sich auf einer DVD, welche der Arbeit beigefügt ist.

5.1 Entwurf

Dieses Kapitel beschreibt den Ablauf der Entwurfs- und Designphase der Master-Portierung. Hierzu wird im ersten Abschnitt zunächst die Ausgangssituation analysiert. Dabei soll vor allem auf die Architektur des IgH EtherCAT-Masters eingegangen werden, um daraus Entscheidungen für das Design der neuen Umsetzung ableiten zu können.

5.1.1 Ausgangssituation

Der dieser Arbeit zugrunde liegende EtherCAT-Master wurde mit dem Ziel entwickelt, ein EtherCAT-Netzwerk mit frei verfügbaren Werkzeugen auf Basis eines Linux-Systems aufbauen zu können. Im Folgenden soll nun die grundlegende Architektur desselben vorgestellt werden, da diese ebenfalls in der Portierung auf DROPS Verwendung findet.

Grundaufbau

Das Grundgerüst des Masters besteht aus drei Kernmodulen:

Mastermodul Dies ist die zentrale Komponente eines EtherCAT-Masters. Sie implementiert die Zustandsmaschine, verwaltet die Prozessdaten und regelt den Ablauf der Kommunikation mit Treiber- und Anwendungsmodul.

Treibermodul Dieses Modul ist für die Kommunikation mit der Netzwerkkarte zuständig. Es handelt sich hierbei um einen angepassten Linux-Netzwerkkartentreiber. Das Master-Modul nutzt später direkt die Sendefunktion des Treibers, um Datenpakete abzuschicken. Der Empfang wird ebenfalls über den Master durch zyklisches Polling ausgelöst.

Anwendungsmodul Diese Komponente nutzt die vom Mastermodul zur Verfügung gestellten Schnittstellen zum zyklischen Austausch von Prozessdaten mit EtherCAT-Slaves.

Ursprünglich wurden alle drei vorgestellten Komponenten als Module im Linux-Kern betrieben. Dies hat zwar den Vorteil, dass die Kommunikation der Echtzeitelemente untereinander schnell und einfach erfolgen kann, erschwert allerdings gleichzeitig den Austausch von Daten mit anderen, im Userspace implementierten, Applikationen (bspw. zur Visualisierung von Daten). Aus diesem Grund wurde in der aktuellen Entwicklungsversion 1.5 damit begonnen, eine Bibliothek zu entwickeln, welche die gesamte EtherCAT-API über ein Zeichengerät (*character device*, e.g. `/dev/EtherCAT0`) zur Verfügung stellt. Benutzerprogramme können diese Bibliothek verwenden und nutzen somit die gewohnte `ecrt_*()`-API. Zusätzlich existiert ein Werkzeug namens *ethercat*, welches für Analyse- und Konfigurationszwecke ebenfalls das angesprochene Zeichengerät verwendet. Hierüber lassen sich bspw. Informationen über die am

EtherCAT-Bus angeschlossenen Slavegeräte abfragen. Abbildung 5.1 veranschaulicht die Architektur des originalen EtherCAT-Masters unter Linux.

Treiberintegration

Um Netzwerkkarten als EtherCAT-Gerät nutzen zu können, sind einige Veränderungen an deren Treibern vorzunehmen. Zum Einen muss sichergestellt werden, dass sich ein Gerät jeweils nur mit einem Netzwerkstack im System - entweder TCP/IP oder EtherCAT - verbindet. Würde dies nicht geschehen, so könnten sich Pakete vermischen und die reibungslose Kommunikation gefährden. Gleichzeitig ist es allerdings möglich, zwei Netzwerkkarten desselben Typs mit demselben Treiber für beide Kommunikationsprotokolle zu verwenden. Die Treiber implementieren zu diesem Zweck bei der Initialisierung eine Entweder-oder-Strategie. Sie versuchen zunächst, als EtherCAT-Gerät akzeptiert zu werden. Schlägt dieses Vorhaben fehl, so registrieren sie sich als gewöhnliches Netzwerkgerät am TCP/IP-Stack des Systems.

EtherCAT-Treiber besitzen eine weitere Besonderheit, die sie von gewöhnlichen Netzwerktreibern unterscheidet. So wird der Empfang eingehender Pakete ausschließlich durch periodisches Abfragen (*polling*) organisiert. Diese müssen hierzu eine geeignete Methode bereitstellen, welche vom EtherCAT-Stack aufgerufen werden kann.

Implementierung

Die Implementierung aller Module im Linux-Kern war eine frühe Designentscheidung der Entwickler. Diese Wahl basierte hauptsächlich auf drei Annahmen (vgl. [57], S. 5):

1. Programme, die im Kernmodus ablaufen, zeigen wesentlich besseres Echtzeitverhalten als solche, die im Benutzermodus ausgeführt werden.
2. Kommunikation in Feldbussen findet typischerweise zyklisch statt und sollte aus diesem Grund im Systemkern implementiert werden. Dabei versprechen weniger Kontextwechsel auch verringerte Latenzzeiten.
3. Netzwerkkartentreiber werden in Linux-Systemen ausschließlich im Kern implementiert. Es bietet sich daher an, Netzwerkprotokolle ebenfalls im Systemkern

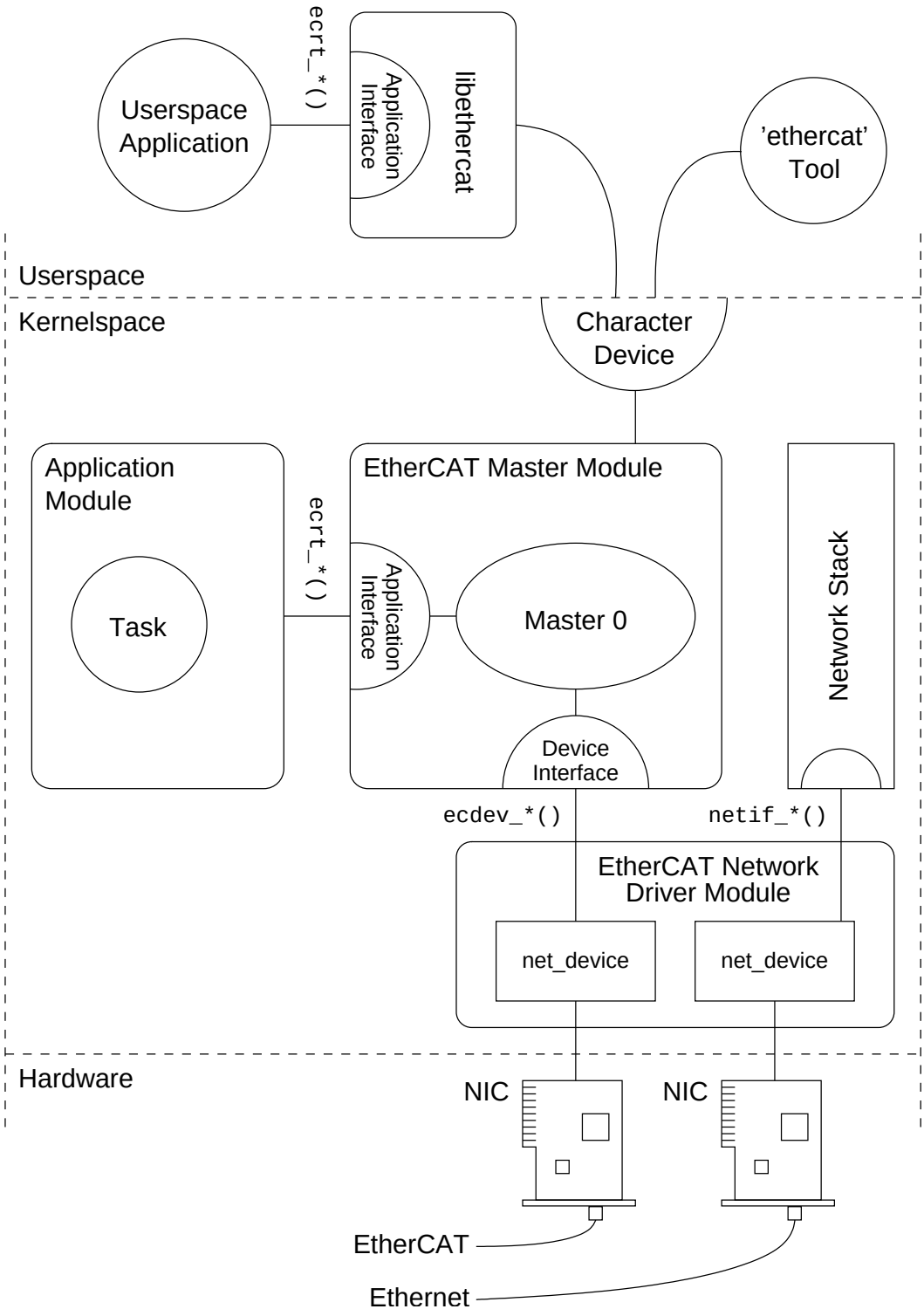


Abb. 5.1: Architektur des originalen EtherCAT-Masters ([57], S. 16)

zu platzieren, da das naturgemäß hohe Kommunikationsaufkommen dadurch besser bewältigt werden kann.

Die angebrachten Argumente sind grundsätzlich korrekt. Dennoch erscheinen die negativen Auswirkungen als zu hoch bewertet. Insbesondere der erste Punkt wurde von den Entwicklern selbst entkräftet. So konnte durch Messungen bspw. gezeigt werden, dass die zeitliche Verzögerung durch die Userspace-Bibliothek im Vergleich zur Kernelimplementierung nur $1\mu s$ pro Funktionsaufruf beträgt ([57], S. 64).

Die Realisierung von EtherCAT-Applikationen im Userspace erleichtert aber nicht ausschließlich die Interaktion mit anderen Anwendungen, auch die Stabilität erhöht sich. Eine fehlerhaft implementierte Anwendung kann nun nicht mehr das gesamte Betriebssystem zum Absturz bringen.

Gerade in sicherheitskritischen Bereichen ist eine hohe Verlässlichkeit aller eingesetzten Komponenten unerlässlich. Logische Konsequenz dieser Tatsache wäre die weitere Aufspaltung und Isolation der Subsysteme voneinander (vgl. Kapitel 4.2). Dieses Vorhaben ist in linuxbasierten Systemen allerdings nicht ohne erheblichen Aufwand möglich. Aus diesem Grund wird im nun folgenden Teil der Arbeit die Portierung des EtherCAT-Masters auf ein mikrokernbasiertes System vorgenommen. Ziel dabei ist die Erhöhung der Ausfallsicherheit des Systems durch Verlagerung aller drei genannten Kernmodule in jeweils getrennte Prozesse. Gegenstand weiterer Betrachtung sollen die Auswirkungen dieser Portierung auf die Echtzeitfähigkeit des Systems sein. Im folgenden Abschnitt werden hierzu Anforderungen aufgestellt, die als Orientierungshilfe für anstehende Schritte dienen.

5.1.2 Anforderungsanalyse

Im Vordergrund der Entwicklung eines mikrokernbasierten EtherCAT-Masterknotens steht die Verbesserung der Ausfallsicherheit im Vergleich zu linuxbasierten Lösungen. Im analytischen Teil der Arbeit wurde gezeigt, dass eine erhöhte Verlässlichkeit nicht kostenfrei erreicht werden kann. Aus diesem Grund muss gerade in Bezug auf das Echtzeitverhalten jeder Portierungsschritt gut überlegt sein, sodass hier keine gravierenden Einbußen hingenommen werden müssen. Technisch neu entwickelte Systeme scheitern zudem häufig daran, dass deren Handhabung im Vergleich zum

Vorgängerprodukt erschwert wurde. Vor diesem Hintergrund sollten folgende Aspekte bei der Realisierung Beachtung finden:

Verlässlichkeit Die Verlässlichkeit des Systems soll zum Einen durch die Isolation der Kernkomponenten, zum Anderen durch eine redundante Auslegung des Treibermoduls erreicht werden. Der zuletzt genannte Punkt ermöglicht den Umgang mit defekter Netzwerkhardware sowie fehlerhaften Gerätetreibern. Wird ein Ausfall festgestellt, so soll automatisch auf ein Ersatzgerät ausgewichen werden.

Zeitverhalten Es ist davon auszugehen, dass die angestrebte Verbesserung der Systemstabilität Einfluss auf das zeitliche Verhalten nimmt. In jedem Fall erhöht sich aufgrund des gesteigerten Kommunikationsaufkommens die erreichbare minimale Zykluszeit (Performanzkriterium). Auswirkungen auf das Echtzeitverhalten können ebenso nicht ausgeschlossen werden. Beide Aspekte sollen jedoch auf einem Niveau gehalten werden, das die Benutzbarkeit des Systems nicht wesentlich einschränkt. Hierbei muss besonders bei der Kommunikation der Komponenten untereinander auf eine zweckmäßige Lösung geachtet werden.

Benutzbarkeit Bei der Portierung soll darauf geachtet werden, dass sich die verfügbare API zur Verwendung des EtherCAT-Masters auch unter der mikrokernbasierten Lösung verwenden lässt. Die Programmierung vereinfacht sich damit erheblich, da der Entwickler gewohnte Funktionsaufrufe verwenden kann. Binärkompatibilität kann jedoch nicht erreicht werden. Weiterhin wird die Wiederverwendung des kommandozeilenbasierten Konfigurationswerkzeugs angestrebt. Des Weiteren soll versucht werden, so viel Quellcode wie möglich unverändert vom EtherCAT-Master zu übernehmen, da dies einen leichteren Umstieg auf weiterentwickelte Versionen ermöglicht.

Ausgehend von diesen Zielen, sollen im nun folgenden Kapitel grundsätzliche Entscheidungen zur Erfüllung derselben erläutert werden. Es muss jedoch angemerkt werden, dass diese im Rahmen der vorliegenden Diplomarbeit aus zeitlichen Gründen nicht alle erfüllt werden konnten.

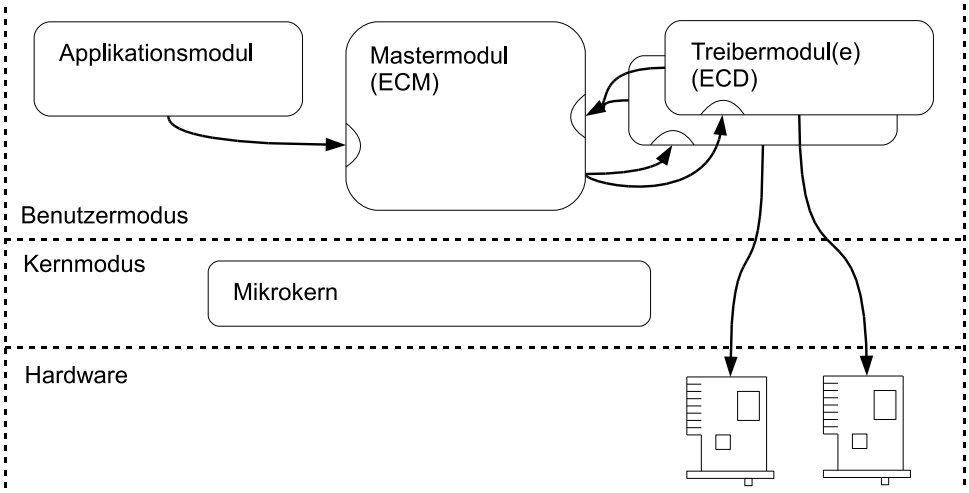


Abb. 5.2: Grundlegende Architektur der EtherCAT-Master Portierung

5.1.3 Entwurfsentscheidungen

Durch die Auslagerung aller Module in getrennte Adressräume werden diese voneinander isoliert. Mastermodul, Treibermodul sowie Anwendungsmodul laufen somit in unterschiedlichen L4-Tasks. Das Mastermodul bildet das Zentrum der Architektur. Es stellt eine Bibliothek zur Verfügung, die Applikationen einbinden und damit die gewohnte EtherCAT-Anwendungs-API nutzen können. Dieser Teil ist somit besonders für Endbenutzer interessant.

Im Allgemeinen muss ein Kommunikationsprotokoll aber auch die Verbindung zur physischen Schicht sicherstellen. In linuxbasierten Systemen verbinden sich hierzu EtherCAT-fähige Netzwerkkartentreiber direkt mit dem Mastermodul und bieten diesem ihre Funktionalität an. Akzeptiert der Master dieses Angebot, so kann nun über den Treiber kommuniziert werden. Da in einer mikrokernbasierten Lösung jedoch beide Komponenten getrennt voneinander implementiert sind, müssen diese über IPC miteinander verbunden werden. Dabei soll ein virtueller Netzwerktreiber zum Einsatz kommen. Dieser registriert sich als einziger Treiber am Mastermodul und setzt somit vollkommen transparent die Kommunikation zwischen diesem und einem oder auch mehreren Treibermodulen um. Hierzu nutzt der virtuelle Netzwerktreiber eine vom Treibermodul zur Verfügung gestellte Bibliothek. Um die Kommunikationskosten niedrig zu halten, sollte aufwendiges Kopieren von Prozessdaten oder ganzen Netzwerkpaketen entfallen. Aus diesem Grund wird vorgesehen, ausschließlich Shared-

Memory zum Nachrichtenaustausch zu gebrauchen. Lediglich die Synchronisation sowie der Aufruf essenzieller Funktionen werden über herkömmliche IPC gesteuert. Abbildung 5.2 verdeutlicht den grundsätzlichen Aufbau des mikrokernbasierten EtherCAT-Masterknotens.

5.2 Implementierung

Aufbauend auf den Vorüberlegungen aus Kapitel 5.1 folgt nun die Beschreibung der Portierung auf das mikrokernbasierte Betriebssystem DROPS. Hierzu werden zunächst beide Grundmodule sowie deren dazugehörige Bibliotheken erläutert. Anschließend erfolgt in Abschnitt 5.2.3 die Vorstellung von zwei Beispielapplikationen. Eine davon, das EtherCAT-Programm *ec_bench*, bildet die Grundlage der Evaluierung, welche im anschließenden Kapitel beschrieben ist.

5.2.1 Mastermodul

Dieser Abschnitt beschreibt Schritte und Details der Implementierung des EtherCAT--Mastermoduls (ECM) unter DROPS. Um eine möglichst hohe Modularität zu gewährleisten, wurden die benötigten Teilkomponenten in eigene Bibliotheken ausgelagert. Bevor diese im Weiteren näher erläutert werden, soll zu Beginn auf das Hauptmodul selbst eingegangen werden. Abbildung 5.3 verdeutlicht zunächst den Gesamtaufbau des Moduls.

Hauptmodul

Das Hauptmodul stellt den Einstiegspunkt der Komponente nach dem Systemstart dar und implementiert die `main()`-Routine. Hauptaufgabe ist die Initialisierung der Ablaufumgebung für alle weiteren Teilsysteme. So wird hier bspw. die Taskpriorität festgelegt und DDE initialisiert. Der zuletzt genannte Schritt sorgt auch dafür, dass die entsprechende Initialisierungsfunktion des EtherCAT-Stacks sowie des virtuellen Netzwerktreibers aufgerufen wird. Sobald der letzte Punkt abgeschlossen ist, registriert sich ECM am Namendienst. Dieser ist mit dem DNS-Dienst (*domain name system*) im Internet vergleichbar und übersetzt den Namen der Applikation in die

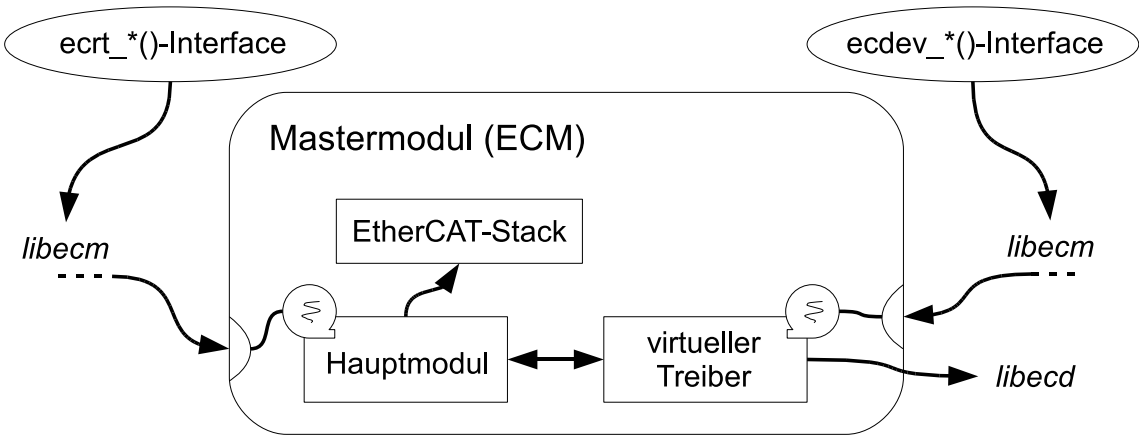


Abb. 5.3: Architektur Mastermodul

aktuelle Task- und Threadidentifikation. Von nun an befindet sich ECM in einer Endlosschleife und wartet auf eingehende Verbindungen, was durch das Threadsymbol in Abbildung 5.3 verdeutlicht wird.

EtherCAT-Stack

Herzstück des Mastermoduls ist der EtherCAT-Stack selbst. Dieser wurde in eine eigene Bibliothek namens *libighmaster* ausgelagert und umfasst sämtliche Quelldateien des originalen EtherCAT-Masters. Zur Einbindung nutzt *ECM* das von *DROPS* zur Verfügung gestellte DDE. DDE exportiert fast die gesamte Funktionalität (API), die eine Komponente auch in einem nativen Linux-System vorfinden würde. Ausnahmen stellen jedoch die Unterstützung von Zeichengeräten, das dynamische Laden von Firmware sowie die Verwendung von Parametern beim Starten von Modulen dar.

Aus diesem Grund mussten für eine erfolgreiche Übersetzung drei Stellen in den Quelldateien `master.c` sowie `module.c` angepasst werden. Diese sind für die Erstellung, Initialisierung bzw. Löschung der Geräteknoten beim Laden und Entladen des Kernelmoduls verantwortlich. Da der Zugriff auf sämtliche Funktionen nun aber über das Kerninterface geschieht, wird das Zeichengerät nicht mehr benötigt. Die Datei `cdev.c` konnte dadurch zum Beispiel komplett vom Übersetzungsvorgang ausgeschlossen werden.

Die zweite Änderung betrifft die Initialisierungsphase des EtherCAT-Stacks selbst.

Ursprünglich wird dem Kernelmodul per Modulparameter eine Liste an Netzwerkkarten (bzw. deren MAC-Adressen) übergeben. Anhand dieser entscheidet der Master, ob eine Netzwerkkarte als EtherCAT-Gerät akzeptiert wird oder nicht. Um das Modul dennoch nutzen zu können, wird die betreffende Variable fest mit der (virtuellen) MAC-Adresse der virtuellen Netzwerkkarte initialisiert.

Virtueller Treiber

Da innerhalb des Mastermoduls kein Zugriff auf native Netzwerkkarten besteht, muss durch einen Treiber ein virtuelles Gerät nachempfunden werden. Treiber dieser Art werden häufig auch als *Stub*-Treiber bezeichnet, da entsprechende Funktionsaufrufe nur emuliert werden. Das im Rahmen dieser Arbeit entwickelte virtuelle EtherCAT-Gerät (*l4ecdev*) wurde ebenfalls in einer eigenen Bibliothek namens *libecdev* implementiert.

Hauptaufgabe des Treibers ist die Umsetzung aller Aufrufe durch den EtherCAT-Stack in entsprechende IPC-Nachrichten für das Treibermodul. Des Weiteren ist hier auch die gesamte Logik für den Zugriff auf verschiedene Instanzen des Treibermoduls implementiert. Im Fehlerfall kann so auf ein Ausweichmodul umgeschaltet werden. Das kann bspw. beim Absturz des Treibers sinnvoll sein. Es sind aber auch Szenarien vorstellbar, die es ermöglichen, den Verlust der physischen Netzverbindung (z. B. Kabel unterbrochen) zu erkennen. Daraufhin könnte zum Beispiel auf eine redundant ausgelegte Zusatzverbindung ausgewichen werden.

Wie bereits erwähnt, fragt der EtherCAT-Stack zum Empfang neuer Pakete periodisch den Treiber ab. Existiert hierfür keine dedizierte Funktion innerhalb des Treibers, so wird diese pragmatisch durch manuellen Aufruf der Interrupt-Service-Routine emuliert. Aufgrund unterschiedlicher Möglichkeiten zur Realisierung von Interrupthandlern in Netzwerkkartentreibern kann die Übergabe der empfangenen Pakete zum EtherCAT-Stack nicht synchron zur Anfrage realisiert werden. Vielmehr rufen Treiber nach Abarbeitung der spezifischen Empfangsroutine eine Callback-Funktion auf.

l4ecdev implementiert zu diesem Zweck einen eigenen Empfangsthread (siehe Abbildung 5.3). Sobald ein EtherCAT-Paket eingegangen ist, wird dies vom Treibermodul durch Aufruf der Funktion `ecdev_receive()` signalisiert. Erst nachdem die Daten

vom Treibermodul zu l4ecdev übertragen wurden, erfolgt die Weiterleitung zum EtherCAT-Stack.

Masterbibliothek

Um das Mastermodul zu nutzen, verwenden Clients die Masterbibliothek (*libecm*). Diese exportiert sowohl das für Anwendungsprogramme interessante `ecrt_*()`-Interface (**E**ther**C**AT **R**eal-**T**ime) als auch Teile des `ecdev_*()`-Interface (**E**ther**C**AT **Dev**ice). Letzteres wird dabei ausschließlich vom Treibermodul genutzt.

Zur Generierung der IPC-Coderoutinen wurde in diesem Projekt ausschließlich *DICE* [58] eingesetzt. DICE ist ein IDL-Compiler (*interface definition language*), welcher Interfacebeschreibungen in ausführbaren Code für L4-kompatible Mikrokerne übersetzt. Der Programmierer braucht sich bei dieser Methode nicht mehr selbst um die recht aufwendige und fehleranfällige Erstellung der Kommunikationsroutinen kümmern.

Hauptziel bei der Entwicklung der Masterbibliothek ist es, das API möglichst wenig zu verändern. Anwender sollen gewohnte Schnittstellen vorfinden, sodass bei einem Umstieg nur wenig Eingewöhnungszeit vonnöten ist. Außerdem kann so derselbe Quellcode für Linux als auch für L4-basierte EtherCAT-Anwendungen verwendet werden. Diese Vorgabe konnte bei den bisher umgesetzten Funktionen erfüllt werden. Dazu zählen die Hauptmethoden zur Steuerung des Masters sowie zur Slave- und Domänenkonfiguration. Damit steht eine Grundausstattung zur Verfügung, mit der einfache EtherCAT-Anwendungen umgesetzt werden können. Die Vervollständigung der Programmierschnittstellen kann aufgrund dieser Beispielfunktionen ohne großen Zeitaufwand durchgeführt werden.

In der aktuell vorliegenden Version ist bei der Initialisierung des Domänenspeichers, der Speicherbereich der zum Datenaustausch zwischen Mastermodul und Applikation dient, eine Besonderheit zu beachten. Bevor der zyklische Betrieb der EtherCAT-Applikation gestartet werden kann, muss zunächst ein geeigneter Datenbereich alloziert und mit entsprechenden Zugriffsrechten für beide Kommunikationspartner versehen werden. Dieser Schritt wird im Moment noch vom Anwendungsmodul durch die neue Funktion `ecrt_domain_malloc()` durchgeführt. Für eine spätere Version ist allerdings die Integration in die `ecrt_master_activate()`-Methode geplant.

Damit wäre dann eine einhundertprozentige Kompatibilität zur vorhandenen API gewährleistet.

5.2.2 Treibermodul

Um tatsächlich Daten in einem EtherCAT-Netzwerk austauschen zu können, benötigt der Master Unterstützung von einem Modul, welches Zugriff auf real existierende Ethernet-Hardware besitzt. An diesem Punkt der Architektur kommt das Treibermodul ins Spiel, welches exakt für diesen Zweck geschaffen wurde. Es beherbergt gewöhnliche EtherCAT-Netzwerktreiber, welche über definierte Schnittstellen mit dem Mastermodul, genauer mit dem virtuellen Netzwerktreiber, kommunizieren. In diesem Abschnitt sollen nun der grundlegende Aufbau des Treibermoduls, die Teilkomponenten sowie Schnittstellen zu anderen Subsystemen erläutert werden. Abbildung 5.4 verdeutlicht zunächst die Architektur des Moduls.

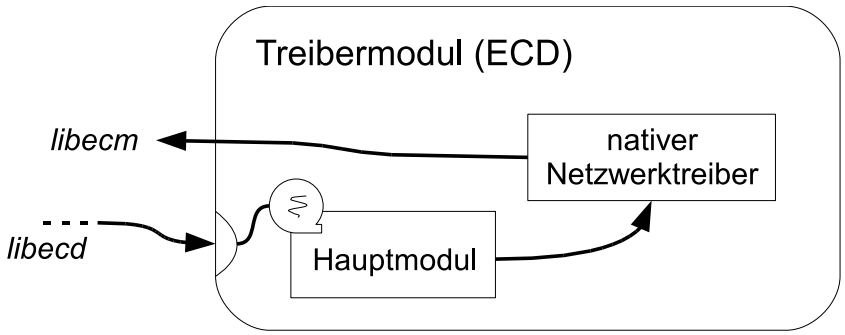


Abb. 5.4: Architektur Treibermodul

Hauptmodul

Im Treibermodul stellt das Hauptmodul ebenfalls den Einstiegspunkt der Komponente nach dem Systemstart dar. Auch hier wird zunächst DDE gestartet, wobei automatisch auch die Initialisierungsfunktion des Netzwerkkartentreibers aufgerufen wird. Nach dessen Start registriert sich das Modul am lokalen Namendienst und tritt im Anschluss daran in eine Endlosschleife ein (Empfangsthread). Von nun an wartet das Hauptmodul auf eingehende Nachrichten. Diese beschränken sich ausschließlich

auf Aufrufe vom Mastermodul und sind in der Quelldatei `server_components.c` implementiert.

Zusätzlich existiert eine weitere Quelldatei namens `ecdev_wrapper.c`. Diese beinhaltet alle Methoden der EtherCAT-Netzwerktreiber, welche nicht durch DDE zur Verfügung gestellt werden und emuliert deren Funktionalität entsprechend.

Nativer Netzwerktreiber

Innerhalb des Treibermoduls wird der Zugriff auf native Netzwerkkarten über die Bibliothek *libnicdriver* realisiert. In dieser Arbeit wurden Fast-Ethernet-Karten vom Typ *Realtek RTL8139* sowie vom Typ *Intel E100* verwendet. Die hierfür verwendeten Treiberdateien (`8139too-2.6.29-ethercat.c` sowie `e100-2.6.29-ethercat.c`) konnten unverändert aus den Quellen des Masterstacks übernommen werden.

Treiberbibliothek

Die Bibliothek eines Servermoduls stellt dessen Funktionalität über einheitliche Schnittstellen zur Verfügung, sodass diese von Clients unkompliziert genutzt werden kann. Beim Treibermodul übernimmt *libecd* diese Aufgabe, welche ausschließlich vom virtuellen Netzwerktreiber des Masters eingebunden wird. Hauptaufgabe dabei ist der Empfang eingetroffener Netzwerkpakete. Des Weiteren ist eine einfache Heartbeat-Funktion integriert. Diese ermöglicht es dem virtuellen Treiber festzustellen, ob das entsprechende Treibermodul noch korrekt arbeitet (und beispielsweise auf Anfragen reagiert). Ist dies nicht der Fall, so kann an dieser Stelle bspw. auf eine Backup-Instanz ausgewichen werden. Funktionen dieser Art werden in Anwendungen mit besonderen Anforderungen an die Systemzuverlässigkeit bzw. Ausfallsicherheit gefordert.

5.2.3 Applikationsmodul

Beide Kernkomponenten - Mastermodul und Treibermodul - sind ohne entsprechende EtherCAT-Applikation obsolet. Zwar werden grundsätzlich nach dem Start beider zuletzt genannten Tasks bereits Daten im Leerlaufmodus (*idle mode*) ausgetauscht, es findet aber keine Aktualisierung der Prozessdaten statt.

Aus diesem Grund wurden im Rahmen dieser Arbeit mehrere Beispielapplikationen entwickelt, welche die Verwendung des Masters demonstrieren und die nachfolgende Evaluierung desselben unterstützen sollen.

ec_abstimeout

Hauptaufgabe von *ec_abstimeout* ist die Demonstration der Initialisierung einer EtherCAT-Verbindung. Darüber hinaus wird gezeigt, wie in L4-basierten Systemen absolute Timeouts zur Generierung eines zyklischen Taktsignals verwendet werden können.

Der Ablauf der Einrichtung unterscheidet sich bis zur Konfiguration des Domänenspeichers nicht von der in Linux. Folgende drei Schritte sind hierfür notwendig:

1. Verbindungsanfrage zum Mastermodul durch `ecrt_request_master()`.
2. Erstellen einer Domäne über `ecrt_master_create_domain()`.
3. Konfiguration der Slaveknoten durch `ecrt_master_slave_config()` bzw. `ecrt_domain_reg_pdo_entry_list()`.

Im nächsten Schritt erfolgt nun die Allokation des gemeinsamen Speichers zum Austausch der Prozessdaten. Unter Linux erfolgt dies für den Benutzer vollkommen transparent in der EtherCAT-Bibliothek. Ziel für die nächste Version ist es, diese Funktionalität ebenfalls in *libecm* zu verbergen. Zurzeit ist jedoch noch der Aufruf einer weiteren Methode, `ecrt_domain_malloc()`, nötig (vgl. Kapitel 5.2.1: Masterbibliothek).

ec_bench

Im Vergleich zu ec_abstimeout ist *ec_bench* ein recht komplexes Beispielprogramm. Dennoch werden auch hier keinerlei aufwendige Regelalgorithmen implementiert. Hauptaufgabe der Applikation ist vielmehr die Analyse des zeitlichen Verhaltens der EtherCAT-Funktionsaufrufe. Diese Funktionsaufrufe sind Teil des Echtzeitpfades der Applikation und haben somit entscheidenden Einfluss auf das Zeitverhalten. Folgende vier Hauptmethoden werden aktuell mit ec_bench untersucht:

1. `ecrt_master_receive()`
   Der Aufruf dieser Funktion bewirkt, dass der EtherCAT-Stack eingetroffene Pakete vom Treibermodul anfordert.
2. `ecrt_domain_process()`
   In dieser Methode wird nun bspw. überprüft, ob im aktuellen Zyklus Pakete von allen Slaveknoten empfangen wurden. Darüber hinaus wird das Prozessabbild entsprechend der Änderungen aktualisiert.
3. `ecrt_domain_queue()`
   Nach dem Empfang der Pakete und der Abarbeitung des Regelalgorithmus wird nun das (veränderte) Prozessabbild für den Versand an alle weiteren Feldbusteilnehmer vorbereitet.
4. `ecrt_master_send()`
   Diese Funktion bildet den Abschluss eines Zyklusses. Nun werden alle Pakete in der Ausgangswarteschlange versendet.

Ziel dieser Analyse ist es, die Mehrkosten der mikrokernbasierten Implementierung im Vergleich zur monolithischen Lösung aufzudecken. Darüber hinaus wird aber auch der zusätzliche Aufwand ersichtlich, welcher unabhängig von der verwendeten Architektur durch die Kommunikation mit dem Netzwerkinterface entsteht. Es wird erwartet, dass die Funktionen eins und vier den größten zeitlichen Anteil in Anspruch nehmen. Beide Domänenfunktionen operieren dagegen lediglich auf lokalen Daten.

Die grundsätzliche Abarbeitungszeit der Funktionen ist durch die Anzahl der Pakete nach oben begrenzt. Es ist zu erwarten, dass diese in mikrokernbasierten Lösungen durch direkte und indirekte Kosten der Kommunikation höher ist.

Um dies experimentell überprüfen zu können, wurden Messpunkte vor und hinter den vier Hauptmethoden installiert. Mithilfe der `rdtsc`-Instruktion (*read timestamp counter*) kann damit präzise die Abarbeitungsdauer der Funktionen gemessen werden. Um für eine statistische Analyse eine relevante Anzahl an Messwerten zu erhalten, kann die Dauer des Experiments als auch der transienten Phase über einen Parameter festgelegt werden.

Alle Messwerte werden in einem Puffer gesammelt und anschließend mit der Bibliothek GSL [59] (*The GNU Scientific Library*) ausgewertet. Alle Messungen wurden nach dem Experiment über eine TCP/IP-Verbindung zu einem weiteren PC zur

Auswertung übermittelt. Auf Letzterem kommt hierzu eine in Python entwickelte Serverapplikation zum Einsatz.

Ein Auszug der Untersuchungsergebnisse ist im nächsten Kapitel dargestellt. Weitere Messdaten bzw. deren grafische Darstellung befinden sich auf der beiliegenden DVD.

5.3 Zusammenfassung

In diesem Kapitel wurde die Portierung eines ursprünglich für Linux entwickelten EtherCAT-Stacks auf das mikrokernbasierte Betriebssystem DROPS beschrieben. Ausgehend von grundsätzlichen Anforderungen, wurde hierfür zunächst ein Konzept entworfen, welches anschließend, wie in Kapitel 5.2 beschrieben, implementiert wurde.

Obwohl zum gegenwärtigen Zeitpunkt die EtherCAT-API noch nicht vollständig umgesetzt ist, kann das Ergebnis dieser Arbeit bereits als einsatzfähiger EtherCAT-Stack für DROPS beschrieben werden. Die Möglichkeit, zwei gleichwertige Netzwerktreiber zu verwenden, von denen einer im Notfall die gesamte Kommunikation übernimmt, ist dagegen ein Alleinstellungsmerkmal dieser Implementierung.

Kapitel 6

6 Leistungsbewertung

In diesem Kapitel soll das Resultat der gerade beschriebenen Portierung eines EtherCAT-Masters auf ein mikrokernbasiertes Betriebssystem evaluiert werden. Eine Vergleichsgrundlage stellt dabei die unmodifizierte Version des Masters sowie eine selbst entwickelte Beispielapplikation unter dem Betriebssystem Linux dar. Bevor im Anschluss ausführlich auf die nichtfunktionalen Eigenschaften *Echtzeitfähigkeit* und *Verlässlichkeit* eingegangen wird, soll zunächst die verwendete Hard- und Softwarekonfiguration vorgestellt werden.

6.1 Systemkonfiguration

Als Testsystem wurde in dieser Arbeit ein PC mit $1,6GHz$ CPU (*AMD Athlon XP 2000+*) und 1 GB DDR-RAM eingesetzt. Der Prozessor hat $256KByte$ Cache und ist auf einem Mainboard vom Typ *ASRock K7S8X* verbaut. Der Rechner ist zusätzlich mit zwei Netzwerkkarten - einer *Realtek RTL8139* und einer *Intel E100* - zur Verbindung mit dem EtherCAT-Netzwerk ausgestattet. Letzteres wird mit dem EtherCAT-Master in der Entwicklungsversion 1.5 (Revision 1824) betrieben. Als Linux-System wurde *Debian* Version 5.0 mit nachträglich installiertem Linux-Kernel Version 2.6.29.5 verwendet. Dieser wurde zusätzlich mit dem *PREEMPT_RT*-Patch [60] in der Version 22 versehen.

Als Vertreter der mikrokernbasierten Architekturen wurde das Betriebssystem DROPS der TU-Dresden eingesetzt [61]. Als Systemtaktquelle wird hier der PIT im pe-

riodischen Modus genutzt. Aufgrund der voreingestellten Granularität von einer Millisekunde sind in der aktuellen Konfiguration keine kürzeren Zykluszeiten möglich. Die Nutzung des APIC-Controllers als Taktquelle wurde ebenfalls im Einzelbetrieb getestet. Hier sind zwar wesentlich kürzere Zykluszeiten durchsetzbar, erste Versuche zeigten aufgrund der experimentellen Implementierung (der Timerkomponente) allerdings stark schwankende Messwerte. In moderneren Systemen kann aber der HPET zur Taktgenerierung für Echtzeittasks verwendet werden. Dieser sollte in der Lage sein, verlässliche Timerereignisse zu erzeugen, ohne gleichzeitig die Schedulerlast zu erhöhen.

Zur Lasterzeugung wurden auf beiden Systemen zwei Applikationen eingesetzt:

Hackbench *Hackbench* wurde ursprünglich zur Evaluierung von Schedulingalgorithmen entwickelt. Es generiert eine große Anzahl an Prozessen, die alle untereinander über Sockets kommunizieren. Dadurch werden Kontextwechsel erzeugt, welche die Löschung des TLB in x86-basierten Prozessoren nach sich ziehen. Detailliertere Informationen zu Hackbench können unter [62] bezogen werden.

Calibrator Dieses Werkzeug wurde ursprünglich entwickelt, um das Speichersystem eines Computers zu analysieren (vgl. [63]). Es erzeugt eine hohe Anzahl an Cache- und TLB-Zugriffen und provoziert damit Fehlzugriffe (*misses*) bei der Ausführung der Echtzeitapplikation.

Beide Programme werden von einem Shellscript in einer Endlosschleife aufgerufen und produzieren somit konstant Systemlast. Das gesamte System befindet sich in einer *Ramdisk* (*initrd*), welche sowohl in Linux als auch unter DROPS[1] als Wurzeldateisystem verwendet wird. Die EtherCAT-Applikation hat in beiden Umgebungen jeweils die höchste Priorität.

6.2 Echtzeitbewertung

Im folgenden Abschnitt wird experimentell das Zeitverhalten von monolithischen sowie mikrokernbasierten Betriebssystemen analysiert. Grundlage hierfür ist die im

[1] in einem paravirtualisierten Linux

Rahmen dieser Arbeit durchgeführte Portierung eines linuxbasierten EtherCAT-Masters. Zur Bewertung sollen drei unterschiedliche Aspekte betrachtet werden. Diese werden nun kurz erläutert und anschließend in eigenen Abschnitten ausführlich besprochen.

Funktionsdauer Der Echtzeitpfad einer EtherCAT-Anwendung besteht typischerweise aus drei Schritten: Daten empfangen, Daten verarbeiten sowie Daten wieder aussenden. Die Verarbeitung der Daten ist applikationsspezifisch und kann aus diesem Grund nicht allgemeingültig betrachtet werden. Der Empfang und Versand von EtherCAT-Paketen ist allerdings obligatorisch[2]. Mithilfe der Beispielapplikation ec_bench soll daher der Einfluss getrennter Adressräume auf das Zeitverhalten der EtherCAT-Methoden analysiert werden.

Zykluszeit In Echtzeitsystemen wird häufig zyklisch eine bestimmte Funktionalität abgearbeitet. Hauptziel dieses Experiments ist es daher zu überprüfen, bis zu welchem Grad ein Echtzeitbetriebssystem in der Lage ist, diese Anforderung, auch unter hoher Systemlast, zu erfüllen. Dazu soll ebenfalls ec_bench eingesetzt werden. Es wird so konfiguriert, dass über eine EtherCAT-Ausgangsklemme ein digitales Rechtecksignal ausgegeben wird. Dieses wird einem Speicheroszilloskop zugeführt und mithilfe von dessen Analysefunktionen statistisch ausgewertet. In einem optimalen System ist die Pulsbreite aller Messwerte identisch.

Latenz Mithilfe dieses Experiments soll über einen längeren Zeitraum die maximale Reaktionszeit auf ein Eingangssignal überprüft werden. Dieses wird von einem externen Gerät generiert und über eine digitale EtherCAT-Eingangsklemme in das System eingespeist. Ähnlich wie im vorangegangenen Beispiel überprüft eine Testapplikation periodisch den Zustand dieser Klemme. Tritt nun eine Veränderung ein, so ist es Aufgabe der Applikation, den neuen Zustand auf einen weiteren Ausgang zu spiegeln. Die zeitliche Differenz zwischen Auftreten des Eingangsimpulses und Änderung des Ausgangssignals ist die Gesamtlatenz des Systems. Diese wird hauptsächlich durch die programmierte Zykluszeit, der Verzögerung durch Hard- und Software im Steuerrechner sowie durch die Ausbreitung der Informationen im Bussystem selbst bestimmt.

[2] die tatsächliche Größe der Prozessdaten ist ebenfalls applikationsspezifisch

6.2.1 Funktionsdauer

Aufgabe der Funktionsdauermessungen ist es, zu überprüfen, ob sich die theoretisch erwartete Erhöhung der Abarbeitungszeit praktisch bewahrheitet. Die Abarbeitungszeit hängt in erster Linie von der Komplexität der durchlaufenen Funktionen ab. Diese ist im Wesentlichen aufgrund der identischen Codebasis gleich, sodass hier keine signifikanten Unterschiede zu erwarten sind. Ein weiterer Punkt, welcher die Funktionsdauer bestimmt, ist die Anzahl der involvierten Komponenten bzw. deren Kommunikationsmuster. In mikrokernbasierten Architekturen sind alle Systemkomponenten in unterschiedlichen Adressräumen implementiert. Der Austausch von Nachrichten ist durch zusätzliche Kontextwechsel daher besonders aufwendig. Bei Funktionen zum Empfang und Versand der EtherCAT-Pakete ist unter DROPS eine deutlich höhere Verzögerung zu erwarten. Dies liegt zum Einen daran, dass gleich drei Module an der Kommunikation beteiligt sind. Darüber hinaus entsteht zusätzlicher Mehraufwand aufgrund der noch nicht optimalen Implementierung. Aktuell werden die Pakete sowohl beim Senden als auch beim Empfangen doppelt kopiert. In einer späteren Version sollen aufwändige Kopieroperationen jedoch durch die gemeinsame Nutzung der Speicherbereiche vermieden werden.

Während der Messungen wurde eine Vielzahl an Daten generiert. Aus Gründen der Übersichtlichkeit sollen an dieser Stelle nur Auszüge präsentiert werden. Dabei fungiert die Methode `ecrt_master_receive()` als Stellvertreter der Funktionen zum Austausch von Netzwerkpaketen. Auf der anderen Seite stehen zwei Funktionen, die jeweils nur auf lokalen Daten operieren und auch nur einmal IPC-Botschaften austauschen. Die Methode `ecrt_domain_process()` soll hier für einen Vergleich herangezogen werden.

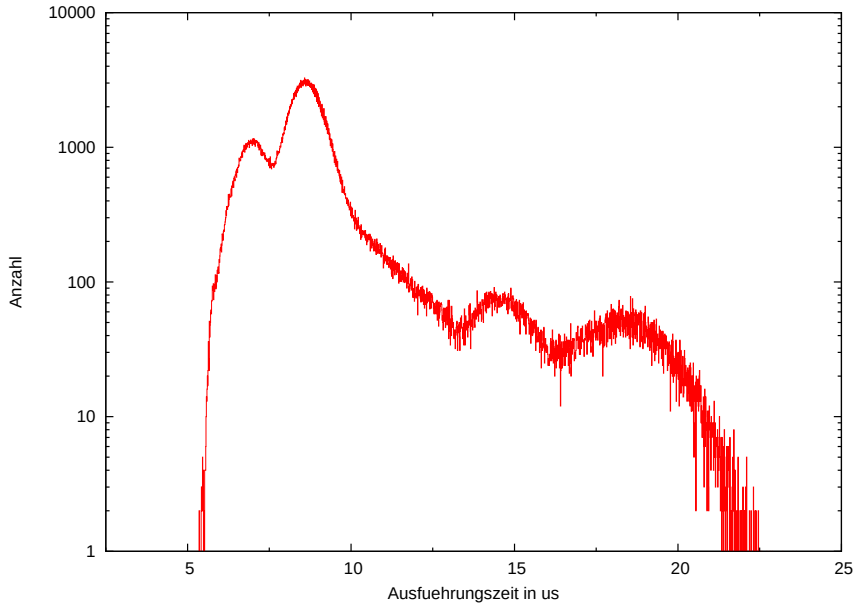
Zusätzlich wurde während der Messungen die Zykluszeit protokolliert. Auch diese wird im vorliegenden Abschnitt diskutiert.

ecrt_master_receive()

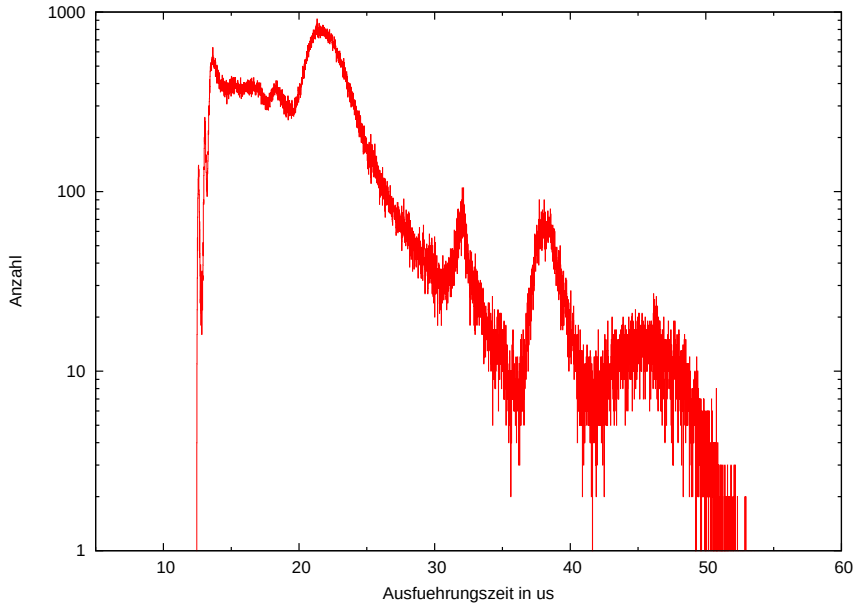
Abbildung 6.1 zeigt das Histogramm der Zeitmessung von `ecrt_master_receive()`. Das Experiment wurde dabei, wie in Abschnitt 6.1 erläutert, aufgebaut. Die Laufzeit betrug zehn Minuten, sodass bei einer Frequenz von $1kHz$ 600.000 Messwerte erzeugt wurden.

Betrachtet man nur die Graphen, so weisen beide Kurven hohe Ähnlichkeiten auf. Die Absolutwerte unterscheiden sich, wie erwartet, jedoch deutlich. So ist die benötigte Zeit zum Empfang eingehender Netzwerkpakete unter DROPS im Mittel $21.1\mu s - 8.9\mu s = 12.2\mu s$ höher als unter Linux. Diese Tatsache lässt sich hauptsächlich durch doppelt kopierte EtherCAT-Pakete erklären. Wie bereits erwähnt, soll das mögliche Optimierungspotenzial in einer der nächsten Versionen ausgeschöpft werden.

Abbildung 6.2 zeigt die Abarbeitungszeit über eine Zeitdauer von zehn Sekunden. Dafür wurde ein Teil der Messdaten als Schnappschuss verwendet. Auffällig ist auch hier eine starke optische Ähnlichkeit der Kurven. Einzeltests mit jeweils nur einem Programm zur Lasterzeugung (vgl. Abschnitt 6.1) haben ergeben, dass sich die starken Ausschläge durch das Programm Hackbench ergeben. Das Werkzeug Calibrator zeichnet sich dagegen vornehmlich für kurze Spitzen verantwortlich, welche bspw. im Bereich zwischen Messwert 8500 ($8.5s$) und 9500 ($9.5s$) beobachtet werden können.



(a) Linux



(b) DROPS

	Minimal	Maximal	Mittelwert	Std. Abw.
Linux	$5.36\mu s$	$23.6\mu s$	$8.9\mu s$	$2.254\mu s$
DROPS	$12.4\mu s$	$55.7\mu s$	$21.4\mu s$	$5.974\mu s$

Abb. 6.1: Histogramm der Funktion `ecrt_master_receive()` in einem belasteten System bei einer Laufzeit von 10 Minuten

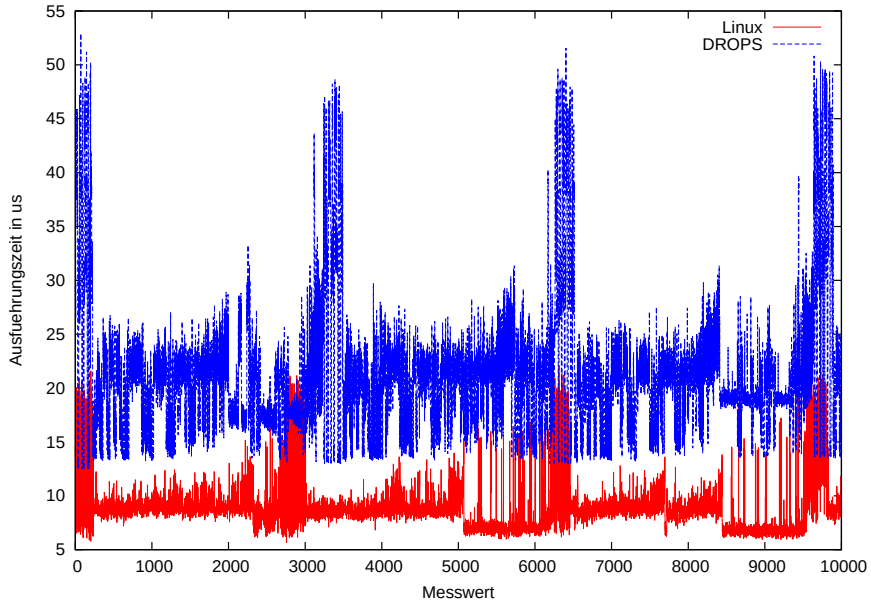
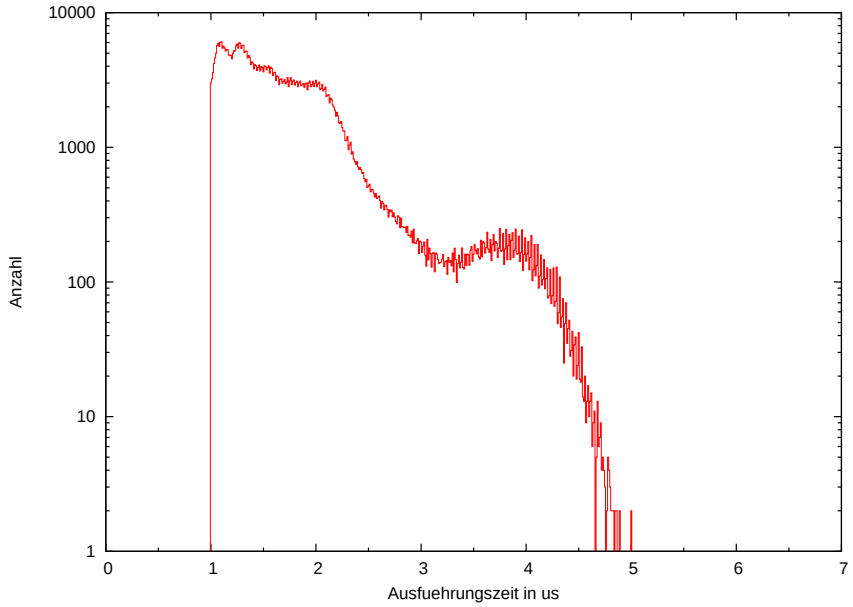


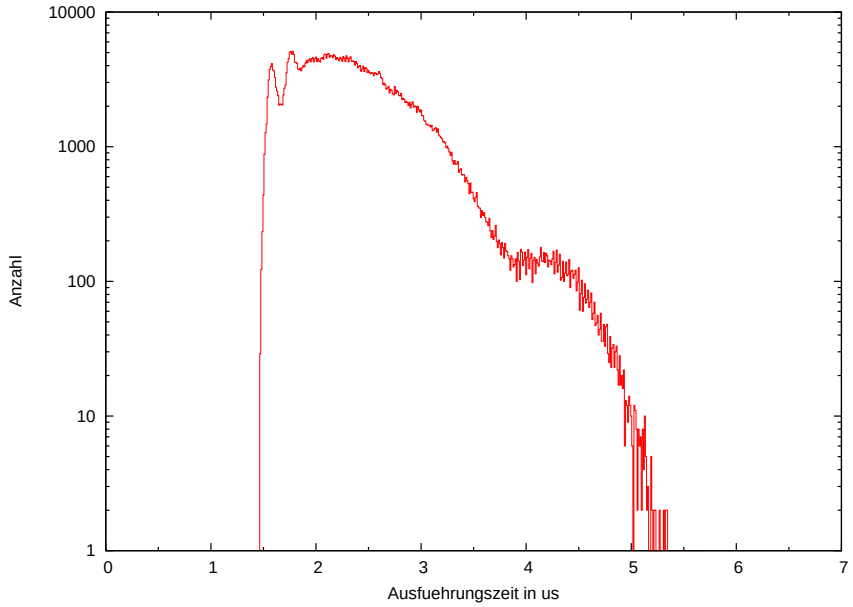
Abb. 6.2: Ausführungszeit der Funktion `ecrt_master_receive()` in einem belasteten System bei einer Laufzeit von 10 Sekunden

ecrt_domain_process()

Die Funktion `ecrt_domain_process()` ist in Bezug auf die anfallende Kommunikation weit weniger aufwändig als die oben betrachtete Methode. Es ist hierfür nur eine IPC-Botschaft nötig (bei der nur die Domänenkennung übertragen wird), sodass Register-IPC ausreicht. Infolgedessen ist das Zeitverhalten beider Funktionen im mikrokernbasierten System, im Vergleich zur nativen Implementierung, nahezu identisch. In Abbildung 6.3 ist lediglich eine Verschiebung des Graphen auf der x-Achse feststellbar. Diese lässt sich durch den nötigen Adressraumwechsel erklären.



(a) Linux



(b) DROPS

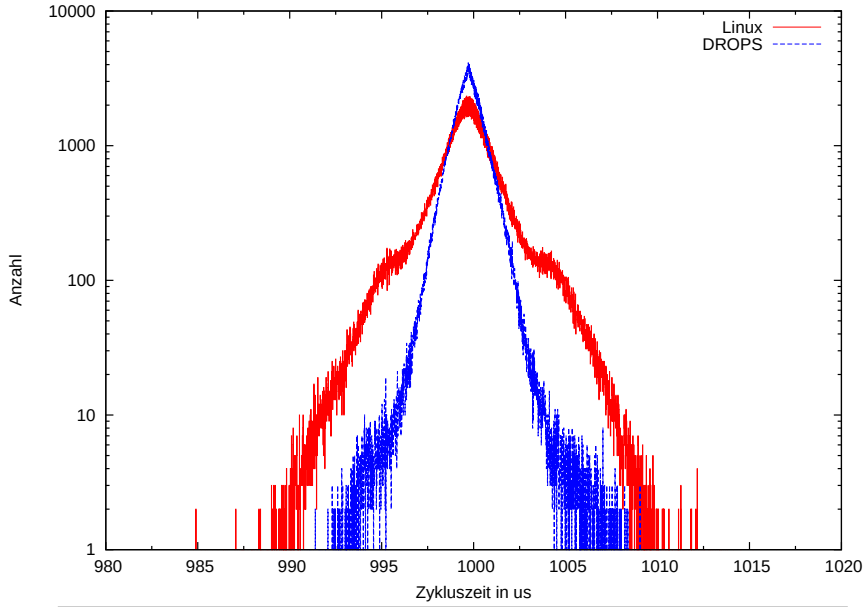
	Minimal	Maximal	Mittelwert	Std. Abw.
Linux	$0.7\mu s$	$4.9\mu s$	$1.6\mu s$	$0.615\mu s$
DROPS	$1.5\mu s$	$5.7\mu s$	$2.4\mu s$	$0.553\mu s$

Abb. 6.3: Histogramm der Funktion `ecrt_domain_process()` in einem belasteten System bei einer Laufzeit von 10 Minuten

Zykluszeit

Die Länge nicht unterbrechbarer Coderoutinen hat entscheidenden Einfluss auf die Echtzeitfähigkeit eines Betriebssystems. Werden im Systemkern zur Synchronisation oder zum Schutz von kritischen Abschnitten stets alle Interrupts abgeschaltet, so beeinträchtigt dies auch das Zeitverhalten von Anwendungen im Userspace. Um Letzteres genauer untersuchen zu können, werden häufig Testprogramme eingesetzt, die periodisch Systemtimer programmieren. Die Abweichung zwischen programmierter und tatsächlich gemessener Periodendauer erlaubt demnach Aussagen über die Unterbrechbarkeit des Systemkerns. Im Umfeld von Linux wird hierzu häufig *cyclictest* [64] genutzt. Das Programm ec_bench bietet eine ähnliche Funktionalität und ist unter Linux sowie unter DROPS einsetzbar. Abbildung 6.4 zeigt ein mit *GNUplot* erstelltes Histogramm der gemessenen Zykluszeit in beiden Systemen. Hierfür wurden absolute Timeouts mit einem Intervall von $1000\mu s$ programmiert.

Auffällig ist auch hier die Ähnlichkeit beider Kurven, wobei der Graph von DROPS eher dem Idealzustand entspricht. Diesen Trend bestätigt auch die statistische Auswertung der Messwerte. Die Achsensymmetrie am Mittelwert ergibt sich durch die Verwendung von absoluten Timeouts. Relative Timeouts würden einen nach links begrenzten Graphen erzeugen, da in diesem Fall nach Abarbeitung der Funktionen immer die zuvor festgelegte Zeit gewartet werden würde.



	Minimal	Maximal	Mittelwert	Std. Abw.
Linux	983.8μs	1016.5μs	999.7μs	2.118μs
DROPS	989.8μs	1010.7μs	999.7μs	0.992μs

Abb. 6.4: Zykluszeit in einem belasteten System bei einer Laufzeit von 10 Minuten

6.2.2 Zykluszeit

In diesem Experiment wird überprüft, ob das jeweils betrachtete Betriebssystem in der Lage ist, einen Echtzeittask unter hoher Systemlast periodisch abzuarbeiten. Die Beispiel-Anwendung *ec_bench* erzeugt dabei ein Rechtecksignal an einer EtherCAT-Ausgangsklemme, welches unter Zuhilfenahme eines digitalen Speicheroszilloskops analysiert wird. Der Echtzeittask wird dabei zyklisch mit einer Frequenz von $1kHz$ aufgerufen, ein optimales System würde daher ein Signal mit einer Pulsbreite von $1000\mu s$ erzeugen. Dieser Versuch ist mit der Bestimmung der Zykluszeit im vorherigen Kapitel vergleichbar. Im Unterschied dazu wird hier jedoch der genaue Zeitwert außerhalb des überprüften Systems bestimmt. Vorteil dieser Methode ist, dass das gesamte System sowie alle Feldbuskomponenten involviert sind.

Abbildung 6.5 verdeutlicht zunächst den Aufbau des Experiments. Im Folgenden sind die gewonnenen Messwerte grafisch dargestellt. Aufgrund der begrenzten Auswertungsfunktionalität des Oszilloskops wurden zur Bildung des Mittelwerts sowie der

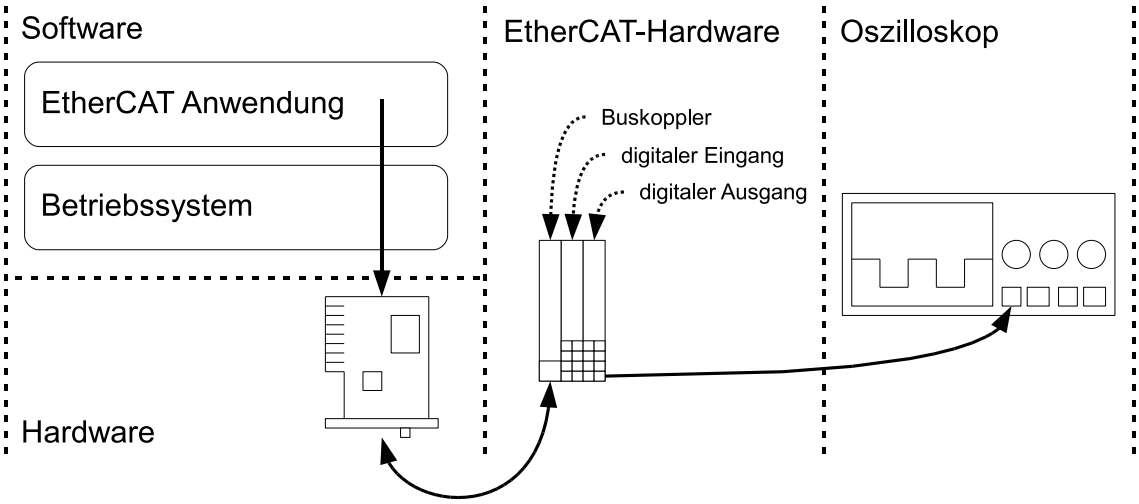


Abb. 6.5: Testaufbau zur Bestimmung der Zykluszeit

Standardabweichung jeweils nur die letzten 1000 Messwerte herangezogen. Minimal- sowie Maximalwerte wurden dagegen über die gesamte Messdauer ermittelt.

Messergebnisse

Abbildung 6.6 zeigt die Ergebnisse der Periodizitätsmessung in einem belasteten und einem unbelasteten Linux-System. Wie erwartet, ist die Differenz zwischen kürzester und längster Pulsbreite (maximaler Jitter) mit $1038\mu s - 961.3\mu s = 76.7\mu s$ in einem belasteten System wesentlich höher als in einem unbelasteten ($1001\mu s - 998.3\mu s = 2.7\mu s$). Wie zu Beginn bereits erwähnt, sind Mittelwert und Standardabweichung nicht aussagekräftig, erlauben allerdings eine Einschätzung der letzten 1000 Messwerte.

In Abbildung 6.7 sind die Ergebnisse der Messungen unter DROPS dargestellt. Auch hier zeigt sich das erwartete Bild: Die Differenz zwischen kürzester und längster Pulsbreite ist im belasteten System mit $82.7\mu s$ deutlich höher als mit $3.7\mu s$ im unbelasteten. Auffällig ist, dass das mikrokernbasierte System im belasteten Zustand bessere Ergebnisse liefert als das monolithische Pendant. Auch die Standardabweichung der letzten 1000 Messwerte belegt diese Tendenz. Im unbelasteten Fall ist die Situation umgekehrt. Hier liefert das linuxbasierte System, im Vergleich zur Testkonfiguration mit L4Linux, bessere Messergebnisse. Dieser Trend ist ebenfalls an der gemessenen Standardabweichung zu erkennen. Wird die Benchmark ohne

	Minimal	Maximal	Jitter	Mittelwert	Std. Abw.
High-Pulsbreite	966.3μs	1038μs	71.7μs	1000μs	8.248μs
Low-Pulsbreite	961.3μs	1036μs	74.7μs	998.4μs	8.191μs

(a) belastetes System

	Minimal	Maximal	Jitter	Mittelwert	Std. Abw.
High-Pulsbreite	998.6μs	1001μs	2.4μs	999.5μs	204.7ns
Low-Pulsbreite	998.3μs	1001μs	2.7μs	1000μs	223.0ns

(b) unbelastetes System

Abb. 6.6: Pulsbreitenmessung unter Linux bei einer Laufzeit von 10 Minuten

	Minimal	Maximal	Jitter	Mittelwert	Std. Abw.
High-Pulsbreite	968.0μs	1045μs	77μs	1000μs	4.711μs
Low-Pulsbreite	962.3μs	1022μs	59.7μs	999.2μs	5.100μs

(a) belastetes System

	Minimal	Maximal	Jitter	Mittelwert	Std. Abw.
High-Pulsbreite	997.3μs	1001μs	3.7μs	999.7μs	448.1ns
Low-Pulsbreite	998.8μs	1001μs	2.2μs	999.9μs	426.3ns

(b) unbelastetes System, mit L4Linux Instanz

	Minimal	Maximal	Jitter	Mittelwert	Std. Abw.
High-Pulsbreite	999.4μs	1001μs	1.6μs	999.8μs	258.7ns
Low-Pulsbreite	998.9μs	1000μs	1.1μs	999.8μs	277.7ns

(c) unbelastetes System, ohne L4Linux Instanz

Abb. 6.7: Pulsbreitenmessung unter DROPS bei einer Laufzeit von 10 Minuten

zusätzliche L4Linux-Instanz gestartet, so ist das mikrokernbasierte System wieder im Vorteil.

6.2.3 Latenz

Die bisher durchgeführten Messungen konnten eindrucksvoll die Unterschiede und Gemeinsamkeiten beim Zusammenspiel der EtherCAT-Implementierung mit beiden betrachteten Betriebssystemarchitekturen belegen. Hierbei wurde neben betriebssystemspezifischer Funktionalität (periodischer Taskaufruf) vor allem das Zeitverhalten einzelner Funktionen betrachtet. Bei der Bewertung des Gesamtsystems zählt

jedoch vielmehr die Interaktion aller Komponenten in einer typischen Arbeitsumgebung. Aus diesem Grund soll im folgenden Abschnitt eine im industriellen Umfeld typische Aufgabenstellung untersucht werden. Dabei wird die Maximallatenz bestimmt, mit der ein Echtzeitrechner auf eine Systemeingabe reagieren kann. Diese Systemanforderung ist zum Beispiel bei Not-Aus-Schaltern in industriellen Anlagen interessant, bei denen bspw. das Signal einer Lichtschranke die Stromzufuhr eines Schrittmotors unterbrechen soll. Es ergibt sich die Frage, ob das verwendete Betriebssystem hier das Reaktionsvermögen der Anlage beeinflusst.

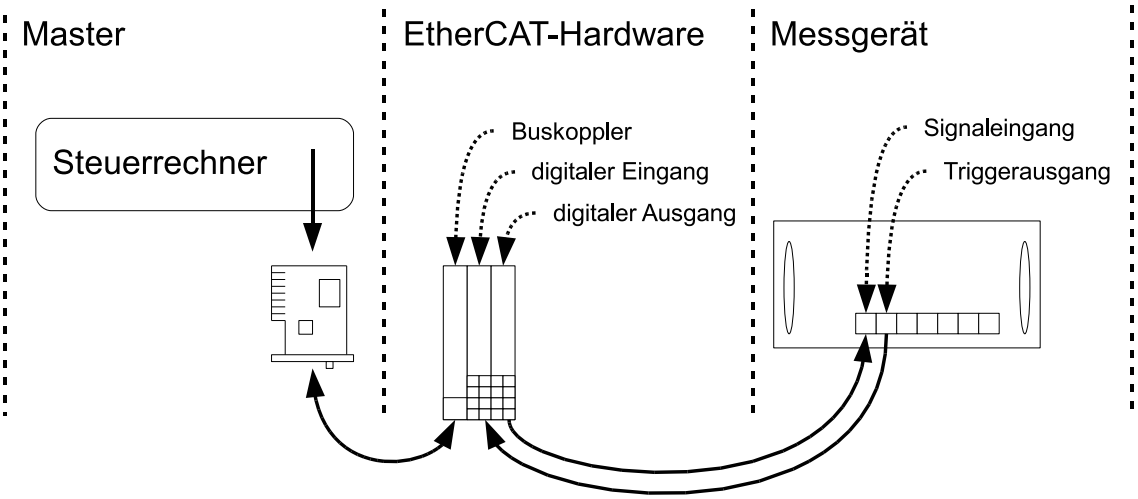


Abb. 6.8: Aufbau des Experiments zur Bestimmung der Latenz

Abbildung 6.8 zeigt den Aufbau eines Experiments, welches dieses Szenario nachahmt. Das Kernstück dabei bildet ein Messgerät [65], was speziell für die experimentelle Bestimmung von Latenzen entwickelt wurde. Es fungiert dabei gleichzeitig als Signalquelle ("Lichtschranke durchbrochen") sowie als Signalsenke ("Motor aus"). Ein FPGA im Messgerät erzeugt, ähnlich wie ein Funktionsgenerator, ein frei parametrierbares Rechtecksignal, welches als *Triggersignal* mit der Eingangsklemme verbunden wird. Der Ausgang des EtherCAT-Moduls ist wiederum mit dem Eingang des Messgeräts verbunden. Dieses ist dadurch in der Lage, die zeitliche Differenz (Latenz) zu bestimmen, die zwischen Ausgabe des Triggersignals bis zum Eintreffen der Systemantwort vergeht.

Im Anschluss an das Experiment können die erhobenen Messwerte[3] über eine Netz-

[3] Ausgabe erfolgt direkt als Histogramm

werkverbindung auf einen externen Rechner übertragen werden. Mithilfe des Messgeräts können Versuchsreihen über mehrere Stunden oder Tage durchgeführt werden, sodass eine große Abdeckung erreicht wird.

Erwartetes Verhalten

Im schlechtesten Fall (*worst case*) tritt das Signal genau dann auf, wenn das EtherCAT-Paket vom aktuellen Zyklus (Buszyklus 1) den Slaveknoten gerade passiert hat. Nun muss zunächst der gesamte Zyklus abgewartet werden. Erst bei der nächsten Abfrage der Eingangsinformationen können diese vom Slaveknoten in das Prozessabbild geschrieben werden (Buszyklus 2). Am Ende dieses Zyklusses befinden sich die Informationen im Master, sodass dieser im nächsten Verarbeitungsschritt (Buszyklus 3) darauf reagieren kann und passende Ausgangsvariablen berechnet. Nachdem das aktuelle Prozessabbild den Masterknoten verlassen hat, können die jeweiligen Belegungen der Slaves noch im aktuellen Takt gesetzt werden. Die Gesamtreaktionszeit beträgt im schlechtesten Fall demnach fast zwei Buszyklen. Im besten Fall sollte die Reaktionszeit nur etwas mehr als einen Buszyklus betragen.

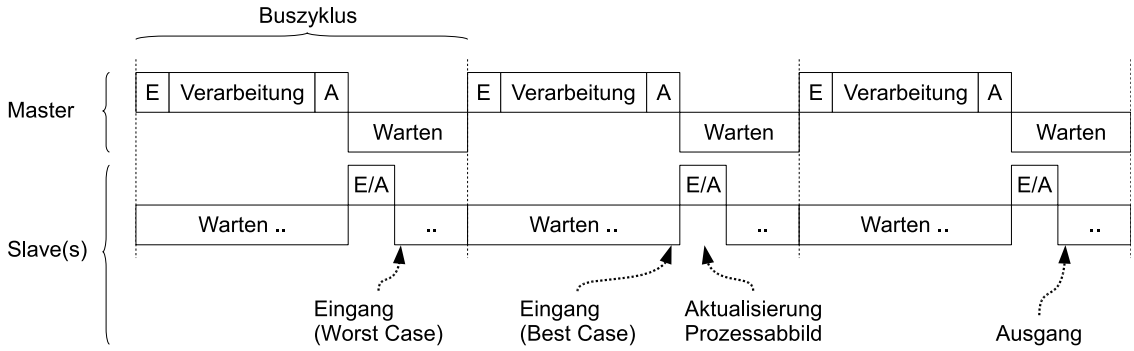


Abb. 6.9: Erwartete EtherCAT-Latenz im besten und schlechtesten Fall (vgl. [23], S. 15)

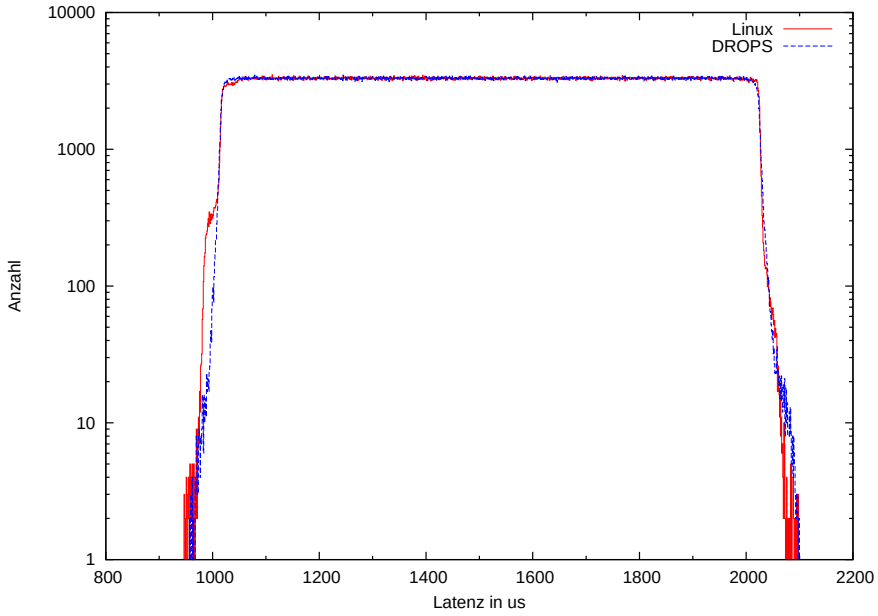
Tatsächliches Verhalten

Im Folgenden soll nun überprüft werden, ob sich die theoretischen Vorüberlegungen in der Praxis tatsächlich bewahrheiten. Hierfür wurde das Experiment wie oben

beschrieben aufgebaut. Die Messungen wurden unter Last in beiden Systemen über eine Laufzeit von 12 Stunden durchgeführt. Die Zykluszeit betrug eine Millisekunde.

Abbildung 6.10 zeigt die erhobenen Daten. Dabei sticht sofort die Ähnlichkeit beider Kurven, sie sind fast deckungsgleich, ins Auge. Die Latenzzeiten liegen im Bereich von 943μs (Minimalwert Linux) und 2102μs (Maximalwert DROPS). Dazwischen verteilen sich die Messwerte gleichmäßigen über den gesamten Zeitbereich. Sie entsprechen damit annähernd exakt den theoretisch ermittelten Sollwerten. Die Abweichungen beim Minimal- und Maximalwert ergeben sich zum Einen durch Schwankungen der Zykluszeit (vgl. Abschnitt 6.2.1: Zykluszeit). Außerdem müssen die Verzögerungen bei der Ausbreitung der Informationen (z. B. Schaltverzögerung an den EtherCAT-Ausgangsklemmen) beachtet werden.

Die geringe Differenz der Maximalwerte beider Systeme zeigt, dass die grundsätzlich länger andauernden Funktionen der mikrokernbasierten Lösung, insbesondere `ecrt_master_receive()`, keinen wesentlichen Einfluss auf die gemessene Latenz haben.



	Minimal	Maximal
Linux	943μs	2099μs
DROPS	958μs	2102μs

Abb. 6.10: Gemessene Latenz unter Linux und DROPS

6.2.4 Fazit

In diesem Kapitel wurden experimentell die Echtzeiteigenschaften in einem monolithischen und einem mikrokernbasierten Betriebssystem unter Zuhilfenahme eines EtherCAT-Stacks analysiert. Um diese bewerten zu können, wurden drei unterschiedliche Aspekte untersucht. Dabei zeigte sich, dass die Kapselung der Komponenten Einfluss auf das zeitliche Verhalten des Gesamtsystems hat. Dies muss allerdings nicht notwendigerweise eine Verschlechterung der Echtzeiteigenschaften bedeuten, was im Experiment zur Bestimmung der Zykluszeit demonstriert wurde. Die Ergebnisse deuten darauf hin, dass die Größe des Systemkerns in direktem Zusammenhang mit dessen Unterbrechbarkeit steht. Die tatsächlichen Mehrkosten einer mikrokernbasierten Lösung hängen entscheidend von der konkreten Anwendung bzw. deren Kommunikationsszenario ab. Im Kontext des hier betrachteten Systems konnte aufgezeigt werden, dass sich diese bei der Verwendung geeigneter Kommunikations- und Synchronisationsmechanismen in einem akzeptablen Rahmen bewegen. Nicht unerwähnt soll dabei die Tatsache bleiben, dass die Feinabstimmung aller Parameter (z. B. Prioritäten) in komplexen Systemen keine triviale Aufgabe ist und deshalb viel Erfahrung erfordert.

6.3 Verlässlichkeitsbewertung

Verlässliche Computersysteme sind Computersysteme, die in der Lage sind, ihre Funktion, auch unter dem Einfluss von Fehlern, bis zu einem gewissen Level zu erfüllen (vgl. Kapitel 2.2). Um dieses Ziel erreichen zu können, muss ein Betriebssystem in der Lage sein, aufgetretene Fehler zu erkennen und geeignete Gegenmaßnahmen zu treffen. Ein möglicher Ansatz zur Fehlerbehandlung ist hierbei der Neustart der ausgefallenen Komponente. Diese Technik wird bspw. in Minix eingesetzt. Ein *Reincarnation-Server* genannter Dienst überwacht dabei stetig den Status anderer Systemservices. Wird ein Ausfall oder Absturz festgestellt, so wird der betroffene Server einfach beendet und neugestartet. Ob der Neustart eines ausgefallenen Subsystems überhaupt sinnvoll ist, hängt von unterschiedlichen Gesichtspunkten ab. Zum Einen können durch einen Neustart nicht alle möglichen Fehler behoben werden, zum Anderen sprengt dieser möglicherweise den zur Verfügung stehenden Zeitrahmen zur Fehlerbehandlung.

Um fehlerhafte Komponenten zu maskieren, muss in verlässlichen Echtzeitsystemen daher häufig eine andere Form der Redundanz gewählt werden. Ein Begriff aus dem Umfeld von hochverfügbaren IT-Systemen ist der sogenannte *Hot-Standby-Modus*. Hierbei werden zwei identische Komponenten parallel betrieben. Eine der Beiden dient dabei als Hauptgerät, während die Andere als Zusatzgerät im Hintergrund mitläuft. Fällt die Primärkomponente aus, so soll die gesamte Funktionalität blitzschnell von der zweiten Instanz übernommen werden.

Bei Netzwerk- und Feldbussystemen nehmen Treiber für die entsprechende Kommunikationshardware eine wichtige Rolle in der Systemarchitektur ein. In Kapitel 4.2 wurden Gerätetreiber aber als eine der Hauptursachen für Systemausfälle identifiziert. Aus diesem Grund ist es gerade für sicherheitskritische Anwendungen unerlässlich, diesen Teil des Systems geeignet zu isolieren und damit das Ausfallrisiko zu minimieren.

Im Folgenden soll daher näher untersucht werden, in welchem Maß in beiden hier betrachteten Betriebssystemen auf mögliche Fehler in Netzwerkkartentreibern reagiert werden kann. Die zuvor beschriebene Technik (Hot-Standby) soll dabei in leicht abgeänderter Form eine Anwendung finden.

6.3.1 Fehlermodell

Eine experimentelle Analyse der Systemverlässlichkeit erfordert zunächst, ein geeignetes Fehlermodell zu definieren. Wie bereits erwähnt, konzentrieren sich die Bemühungen dieser Arbeit auf die Behandlung von Fail-Stop-Ausfällen.

In Kapitel 4.2.2 wurden vier Faktoren vorgestellt, die bei dem Entwurf von verlässlichen Computersystemen unbedingt berücksichtigt werden sollten. Ein Teilaspekt dabei ist die Nutzung der Ressource CPU. Wird diese nicht kontrolliert, so könnte ein fehlerhafter Gerätetreiber bspw. kritische CPU-Instruktionen ausführen oder unendlich viel Prozessorzeit in Anspruch nehmen.

Der Umfang heutiger Netzwerktreiber ist enorm und überschreitet zum Teil den kleinerer Betriebssystemkerne. Zusätzlich sind viele Treiber schlecht dokumentiert, häufig fehlt zudem jegliche Dokumentation der Hardware. Vor diesem Hintergrund sind oben erwähnte Fehler durchaus denkbar. Deren Auswirkungen auf das betroffene System sollen daher nun experimentell analysiert werden. Programmfragment 6.1 zeigt dazu einen beispielhaft in die Senderoutine der Netzwerktreiber injizierten Fehler.

Listing 6.1: Injizierter Fehler in Senderoutine

```
static int rtl8139_start_xmit ( struct sk_buff *skb,
                                struct net_device *dev)
{
        ..
        static int send_counter = 16000;
        if (!send_counter--) {
            __asm__("cli\n"
                    "x1:\n"
                    "jmp x1\n");
        }
        ..
}
```

Nach 16.000 versendeten Paketen betritt der Treiber eine Endlosschleife und blockiert die CPU. Die gesamte (EtherCAT-)Kommunikation ist nun gestoppt, da Pakete

weder empfangen noch gesendet werden können. Dies ist in verlässlichen Computersystemen eine kritische Situation, da die korrekte Funktionalität ohne geeignete Fehlerbehandlung nicht gewährleistet werden kann. Zusätzlich wird vor Betreten der Endlosschleife versucht, alle Interrupts am System zu deaktivieren.

6.3.2 Fehlerbehandlung

Ziel der durchgeführten Portierung war es, die drei Hauptmodule (Master, Treiber und Anwendung) in einem jeweils eigenen Adressraum zu implementieren. Vorteil dabei ist die Isolation der Kernkomponenten. Der Ausfall einer einzigen Komponente gefährdet damit ebenso wenig die Systemstabilität wie kritische CPU-Instruktionen.

Die Aufrechterhaltung der Funktionalität kann allerdings auch hier nur unter Zuhilfenahme geeigneter Fehlerbehandlungstechniken gewährleistet werden. Aufgrund der kombinierten Systemanforderung (Verlässlichkeit und Echtzeitfähigkeit) werden an diese besonders hohe Ansprüche gestellt. Ein Neustart des fehlerhaften Netzwerktreibers kommt im Kontext dieser Anwendung aus zeitlichen Gründen nicht infrage.

Der mikrokernbasierte EtherCAT-Stack wurde daher so modifiziert, dass dieser im Fall eines Ausfalls des Haupttreibers auf eine zweite Instanz ausweichen kann. Sie läuft im Normalbetrieb vollkommen transparent im Hintergrund, ist aber im Notfall in der Lage, die ausgefallene Komponente zu ersetzen (Hot-Standby). Die hierfür notwendige Logik ist im virtuellen EtherCAT-Treiber implementiert. Details zu diesem Punkt sind im Kapitel 5.2.2 zu finden.

6.3.3 Bewertung

Um das Systemverhalten zu bewerten, wurden die Netzwerktreiber für beide Architekturen, wie bereits oben beschrieben, modifiziert. Der folgende Abschnitt schildert die Beobachtungen.

Linux

Die Verwendung des geänderten Treibers unter Linux zeigte das erwartete Verhalten. Da der gesamte Code im privilegierten Prozessormodus ausgeführt wird, kann der Treiber problemlos den CPU-Befehl zur Sperrung der Interrupts ausführen. Das System ist in dieser kritischen Phase darauf angewiesen, dass die Interrupts im weiteren Programmablauf wieder eingeschaltet werden. Geschieht das nicht, so sind alle Mechanismen zur Systemsynchronisation (z. B. periodischer Schedulertakt) wirkungslos.

Im vorliegenden Codebeispiel wird nach der Sperrung der Interrupts eine Endlosschleife betreten. Das daraus folgende Resultat (der sofortige Systemstillstand) kann in dieser Situation nur noch durch ein Systemreset unterbrochen werden.

DROPS

Unter dem mikrokernbasierten Betriebssystem DROPS laufen Gerätetreiber prinzipiell im Userspace. Sie sind daher nicht von gewöhnlichen Applikationen zu unterscheiden, sodass die Ausführung von privilegierten CPU-Instruktionen nicht gestattet ist. Konkret bedeutet das, dass der zuständige Ressourcenmanager (*L4RM*) einen Fehler (*exception*) meldet, sobald ein laufender Thread dies versucht. Die Ausführung des Threads wird daraufhin vom Kern gestoppt[4], sodass ab diesem Moment auch hier keine EtherCAT-Kommunikation mehr stattfindet.

Im Unterschied zu Linux können dessen ungeachtet geeignete Maßnahmen getroffen werden, um die Systemfunktion wiederherzustellen.

Im vorliegenden Fall wurden sämtliche Aufrufe des Treibermoduls (durch den virtuellen Netzwerktreiber) mit einem Timeout versehen. Die Ausführung des Treibermoduls wurde gestoppt (inszenierter Absturz). Diese Tatsache kann jedoch vom virtuellen Treiber erkannt werden, da der Systemkern in diesem Fall eine entsprechende Fehlermeldung liefert. Der virtuelle Treiber ist somit in der Lage, die gerade gestellte Anfrage noch einmal an eine Ausweichinstanz (Backup) zu senden. Ist dieser Versuch erfolgreich, so kann nun jede weitere Kommunikation über den zweiten Netzwerktreiber erfolgen. Für die EtherCAT-Anwendung selbst erfolgt die Umschaltung zwischen

[4] Interrupts bleiben danach eingeschaltet

beiden Treibermodulen ohne Einschränkungen. Es soll jedoch auch darauf hingewiesen werden, dass sich im Fehlerfall die Abarbeitung der gerade ausgeführten Operation zeitlich verzögert. Für Letztere ist weniger der Umschaltvorgang selbst verantwortlich, sondern vielmehr die Zeit, die vergeht, bis der Ausfall einer Komponente erkannt werden kann[5].

6.3.4 Fazit

Fehlerhafte Gerätesoft- und Hardware zeichnet sich für einen Großteil der Ausfälle in Computersystemen verantwortlich. Der potenziell verursachte Schaden hängt maßgeblich davon ab, mit welchen Systemrechten die fehlerhafte Komponente ausgestattet ist. Monolithische Betriebssysteme weisen hier klare Nachteile auf, da die gesamte Funktionalität (also auch die Gerätetreiber) im Systemkern implementiert ist.

Mikrokernbasierte Betriebssysteme konnten als ein möglicher Lösungsansatz identifiziert werden. Diese Annahme wurde im Kontext der hier verwendeten EtherCAT-Umgebung bestätigt. Durch konsequente Isolation bzw. Verlagerung der Gerätetreiber in eigene virtuelle Adressräume kann die Ausfallsicherheit eines Computersystems erheblich verbessert werden. Konkret bedeutet dies, dass die mikrokernbasierte Implementierung unter Zuhilfenahme geeigneter Fehlerbehandlungsmechanismen in der Lage ist, einen Fail-Stop-Ausfall des Netzwerktreibers korrekt zu erkennen und fristgerecht zu beheben.

[5] die aktuelle Version sieht eine maximale Sendezeit von 200μs vor

Kapitel 7

7 Zusammenfassung und Ausblick

Beim Einsatz in industriellen Anwendungen werden an Computersysteme hohe Ansprüche gestellt. Funktionale und zeitliche Anforderungen sollen möglichst ohne Einschränkungen, auch unter dem Einfluss von Fehlern, erfüllt werden.

Die vorliegende Arbeit beschäftigte sich mit der Frage, ob mikrokernbasierte Betriebssysteme in der Lage sind, diese kombinierten Herausforderungen zu bewältigen. Es konnte gezeigt werden, dass Systeme, denen dieses Architekturprinzip zugrunde liegt, dessen besondere Vorzüge im Bereich der Verlässlichkeit auch beim Einsatz in industriellen Anwendungen nutzen können. Die Ausbreitung von Fehlern wird durch konsequente Kapselung der Komponenten eingeschränkt. Der größte Teil des Betriebssystems ist im unprivilegierten Userspace implementiert, was von vornherein die Abarbeitung potenziell gefährlicher Operationen eindämmt. Mikrokernbasierte Systeme mindern darüber hinaus die Entstehung von Fehlern. Ein schlanker Betriebssystemkern fördert den Aufbau flexibler und modularer Systeme, was klare Vorteile bei der Softwarequalitätssicherung (zum Beispiel durch Softwaretests) bietet. Hieraus ergibt sich, dass eine Erhöhung der Systemfunktionalität nicht automatisch Einbußen bei der Softwarekomplexität- oder Qualität impliziert.

Zunächst lag die Befürchtung nah, dass die beschriebenen Vorzüge nur zulasten der Echtzeitfähigkeit erreicht werden könnten. Tatsächlich hängen die direkten und indirekten Kosten sehr stark vom konkreten Anwendungsszenario ab. Im Kontext der hier durchgeführten Analyse und Evaluierung hat die bereits erwähnte Verringerung der Komplexität des Systemkerns durchaus positive Auswirkungen auf das

Echtzeitverhalten des Gesamtsystems. Die im Rahmen der Arbeit durchgeführte Portierung eines EtherCAT-Masters konnte wesentlich zu dieser Erkenntnis beitragen. Obwohl noch nicht sämtliche Module der umfangreichen API verfügbar sind, ist das System bereits jetzt einsatzfähig und bietet darüber hinaus mit der Tolerierung von Fail-Stop-Ausfällen eine Funktionalität, die selbst in der Ausgangsimplementierung nicht zu finden ist.

Mikrokernbasierte Betriebssysteme stellen somit tatsächlich eine Möglichkeit dar, verlässliche Systeme mit engen zeitlichen Vorgaben zu entwerfen. Spielraum für Weiterentwicklungen ergibt sich durch die bereits erwähnte Vervollständigung der Schnittstellen sowie durch Performanzoptimierungen beim Empfang und Versand der Netzwerkpakete. Außerdem wird der Einsatz auf anderen Hardwareplattformen, bspw. auf ARM-Architekturen angestrebt. Die Verfügbarkeit einer EtherCAT-Lösung auf diesen Systemen hat das Potenzial, die Stärken von mikrokernbasierten Architekturen hervorzuheben und somit deren Verbreitung im industriellen Umfeld zu forcieren. Hieraus ergeben sich neue Perspektiven für Entwickler, da die Architektur den Spagat zwischen Echtzeitfähigkeit und Verlässlichkeit ermöglicht. Darüber hinaus unterstützen mikrokernbasierte Betriebssysteme die Synthese mehrerer Anwendungen, was wiederum Entwicklungs- und Herstellungskosten spart, den Energieverbrauch verringert und damit die Umwelt schont.

Literaturverzeichnis

[1] Christian Helmuth, Andreas Westfeld, Michael Sobirey. μSINA - Eine mikrokernbasierte Systemarchitektur für sichere Systemkomponenten. In *8. Deutscher IT-Sicherheitskongress des BSI*, Seite 439–453. Bundesamt für Sicherheit in der Informationstechnik, 2003.

[2] Hermann Kopetz. *Real-Time Systems: Design Principles for Distributed Embedded Applications*. Kluwer Acadamic Publishers, 1997. ISBN: 0792398947.

[3] Nicholas Mc Guire, Peter Okech, Georg Schiesser. Analysis of inherent randomness of the Linux kernel. In *Proceedings of the 11th Real-Time Linux Workshop, Dresden*, 2009.

[4] M. Masmano, I. Ripoll, A. Crespo, and J. Real. TLSF: A New Dynamic Memory Allocator for Real-Time Systems. In *16th Euromicro Conference on Real-Time Systems*, Seite 79–88, Catania, Italien, Juli 2004.

[5] Jochen Liedtke, Hermann Haertig, Michael Hohmuth. OS-Controlled Cache Predictability for Real-Time Systems. In *3rd IEEE Real-time Technology and Applications Symposium (RTAS)*, Seite 213–223, 1997.

[6] Richard Barry. FreeRTOS. Internetseite. Abrufbar unter `http://www.freertos.org/`; zuletzt besucht am 17. September 2009.

[7] Algirdas Avizienis, Jean-Claude Laprie and Brian Randell. Fundamental Concepts of Dependability. PDF-Dokument, 2001. Abrufbar unter `http://www.cert.org/research/isw/isw2000/papers/56.pdf`; zuletzt besucht am 22. Juni 2009.

[8] Vaclav Mikolasek. Dependability and Robustness: State of the Art and Challenges. *Workshop on Software Technologies for Future Dependable Distributed Systems, Tokyo*, März 2009.

[9] Alfredo Benso, Raolo Prinetto (Editoren). *Fault Injection Techniques And Tools For Embedded Systems Reliability Evaluation.* Kluwer Acadamic Publishers, 2003. ISBN: 1402075898.

[10] Mario Holbe, TU-Ilmenau. Robustheit in verteilten Systemen: Einführungskapitel. Vorlesungsfolien. Abrufbar unter `http://tu-ilmenau.de/fakia/fileadmin/template/startIA/vsbs/lehre/SS_2009/Robustheit_in_verteilten_Systemen/Folien/kap1.pdf`; zuletzt besucht am 30. Oktober 2009.

[11] David, Chan, Carlyle, Campbell. CuriOS: Improving Reliability through Operating System Structure. In *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, San Diego, CA, Dezember 2008.

[12] Tanenbaum, Herder, Bos. Can we make operating systems reliable and secure? *Computer*, 39(5):44–51, Mai 2006.

[13] Pratt, Fraser, Hand, Limpach, Warfield. Xen 3.0 and the Art of Virtualization. In *Proceedings of the Linux Symposium*, Band 2, Ottawa, Ontario, Canada, Juli 2005.

[14] VMware, Inc. Virtualization Overview (Whitepaper). PDF-Dokument. Abrufbar unter `http://www.vmware.com/pdf/virtualization.pdf`; zuletzt besucht am 2. November 2009.

[15] Sun Microsystems, Inc. VirtualBox. Internetseite. Abrufbar unter `http://www.virtualbox.org`; zuletzt besucht am 2. November 2009.

[16] Accetta, Baron, Bolosky, Golub, Rashid, Tevanian, Young. Mach: A New Kernel Foundation for UNIX Development. In *Proceedings of the USENIX Summer Conference, Altanta, GA, USA*, Seite 93–112, 1986.

[17] Jorrit N. Herder. Towards A True Microkernel Operating System. Masterarbeit, Freie Universität Amsterdam, Februar 2005.

[18] Universität Karlsruhe, Lehrstuhl Systemarchitektur. L4hq Home of the L4 community. Internetseite. Abrufbar unter `http://l4hq.org/`; zuletzt besucht am 2. November 2009.

[19] Manuel Fähndrich, Mark Aiken, Chris Hawblitzel, Orion Hodson, Galen C. Hunt, James R. Larus, Steven Levi. Language support for fast and reliable message-based communication in Singularity OS. In *Proceedings of the 2006 EuroSys Conference*, Seite 177–190, 2006.

[20] Jochen Liedtke. On micro-kernel construction. In *Proceedings of the 15th ACM Symposium on Operating System Principles (SOSP)*, Seite 237–250, 1995.

[21] Wikipedia, The Free Encyclopedia. Microkernel. Internetseite. Abrufbar unter `http://en.wikipedia.org/wiki/Microkernel`; zuletzt besucht am 2. November 2009.

[22] Michael Hohmuth. The Fiasco Kernel: Requirements Definition. Abrufbar unter `http://os.inf.tu-dresden.de/papers_ps/fiasco-spec.ps.gz`; zuletzt besucht am 18. Juni 2009.

[23] EtherCAT Technology Group. Offizielle EtherCAT Einführung als Präsentation. Technischer Bericht, EtherCAT Technology Group, 2009. Abrufbar unter `http://www.ethercat.org/pdf/german/EtherCAT_Einfuehrung_0905.pdf`; zuletzt besucht am 19. Oktober 2009.

[24] Ingenieurgemeinschaft IgH. EtherLab, Ein Open Source Toolkit für die Echtzeit-Code-Generierung unter Linux mit Einbindung der Simulink/RTW- und EtherCAT-Technologie. Internetseite. Abrufbar unter `http://www.etherlab.org`; zuletzt besucht am 19. Juni 2009.

[25] Engel, Freisleben. Autonomic Network Services on a Microkernel. In *Proceedings of EUROCON 2005, Belgrade, Serbia*, Seite 636–639, 2005.

[26] Swift, Annamalai, Bershad, Levy. Recovering device drivers. *ACM Transactions on Computer Systems (TOCS)*, 24(4):333–360, 2006.

[27] OSEKtime: Time-Triggered Operating System Spezifikation V1.0. PDF-Dokument. Abrufbar unter `http://portal.osek-vdx.org/files/pdf/specs/ttos10.pdf`; zuletzt besucht am 25. August 2009.

[28] Andrew S. Tanenbaum. *Moderne Betriebssysteme*. Prentice Hall, 2. Ausgabe, 2002. ISBN: 3827370191.

[29] Wilhelm Haas. OSEK/OSEKtime OS. Hauptseminarbeitrag, PDF-Dokument. Abrufbar unter `http://www4.informatik.uni-erlangen.de/`

`Lehre/SS06/HS_AKES/handout/OSEK.pdf`; zuletzt besucht am 29. September 2009.

[30] Andrew S. Tanenbaum. *Modern Operating Systems*. Prentice Hall, 3. Ausgabe, 2008.

[31] Hohmuth, Härtig. Pragmatic Nonblocking Synchronization for Real-Time Systems. In *Proceedings of the USENIX Annual Technical Conference*, Seite 217–230, Berkeley, CA, USA, 2001.

[32] Betriebssysteme-Gruppe der TU-Dresden. Microkernel-Based Operating Systems, Kapitel: Real-Time. Vorlesungsfolien. Abrufbar unter `http://os.inf.tu-dresden.de/Studium/KMB/WS2008/05-Real-Time.pdf`; zuletzt besucht am 4. September 2009.

[33] Johannes Plötner, Steffen Wendzel. *Linux - Das distributionsunabhängige Handbuch*. Galileo Computing, 2. Ausgabe, 2007. ISBN: 978-3-8362-1090-4.

[34] Dozio, Mantegazza. Linux Real Time Application Interface (RTAI) in low cost high performance motion control. In *Proceedings of Motion Control 2003*, Mailand, Italien, März 2003.

[35] Fiasco/L4 System Call C-Bindings Reference Manual. Internetseite. Abrufbar unter `http://os.inf.tu-dresden.de/l4env/doc/html/l4sys-l4v2/index.html`; zuletzt besucht am 4. September 2009.

[36] Jork Löser, Hermann Härtig, Lars Reuther. A Streaming Interface for Real-Time Interprocess Communication. Technischer Bericht, TU-Dresden, 2001.

[37] Jane W. S. Liu. *Real-Time Systems*. Prentice-Hall, Inc., 2000. ISBN: 0-13-099651-3.

[38] Sergio Ruocco. A Real-Time Programmer's Tour of General-Purpose L4 Microkernels. Hindawi Publishing Corporation, 2008.

[39] Jork Löser, Michael Hohmuth. Omega0: A portable interface to interrupt hardware for L4 systems. In *Proceedings of the 1st Workshop on Common Microkernel System Platforms*, Seite 200–210, 2000.

[40] Betriebssysteme-Gruppe der TU-Dresden. Microkernel-Based Operating Systems, Kapitel: Device Drivers. Vorlesungsfolien. Abrufbar unter `http://os.`

`inf.tu-dresden.de/Studium/KMB/WS2008/06-Drivers.pdf`; zuletzt besucht am 13. August 2009.

[41] Frank Mehnert, Michael Hohmuth, Hermann Härtig. Cost and Benefit of Separate Address Spaces in Real-Time Operating Systems. In *23rd IEEE Real-Time Systems Symposium (RTSS)*, Seite 124–133, Austin, Texas, USA, 2002.

[42] Douglas Lea. A Memory Allocator. Internetseite. Abrufbar unter `http://gee.cs.oswego.edu/dl/html/malloc.html`; zuletzt besucht am 18. August 2009.

[43] XiaoHui Sun, Xiao Chen, JinLin Wang. An Improvement of TLSF Algorithm. 2007. Abrufbar unter `http://conferences.fnal.gov/cgi-bin/rt2007/download.pl?paper_id=PS1A006&wanted_file=PS1A006.PDF`; zuletzt besucht am 18. August 2009.

[44] Betriebssysteme-Gruppe der TU-Dresden. Microkernel-Based Operating Systems, Kapitel: Memory. Vorlesungsfolien. Abrufbar unter `http://os.inf.tu-dresden.de/Studium/KMB/WS2008/03-Memory.pdf`; zuletzt besucht am 18. August 2009.

[45] Raimund Kirner, Peter Puschner. Classification of WCET Analysis Techniques. In *Proceedings of the 8th IEEE International Symposium on Object-oriented Real-time distributed Computing*, Seite 190–199, Mai 2005.

[46] Chou, Yang, Chelf, Hallem, Engler. An empirical study of operating systems errors. In *Proceedings of the 18th ACM symposium on Operating systems principles (SOSP)*, Seite 73–88, 2001.

[47] Herder, Bos, Gras, Homburg, Tanenbaum. Fault Isolation for Device Drivers. In *Proceedings of the 39th International Conference on Dependable Systems and Networks (DSN)*, Seite 33–42, 2009.

[48] Jeff Arnold, M. Frans Kaashoek. Ksplice: Automatic Rebootless Kernel Updates. In *Proceedings of the 2009 ACM SIGOPS EuroSys Conference on Computer Systems*, April 2009.

[49] Wikipedia, The Free Encyclopedia. Softwarequalität. Internetseite. Abrufbar unter `http://de.wikipedia.org/wiki/Softwarequalität`; zuletzt besucht am 2. September 2009.

[50] Jeremy Miller. Muster in der Praxis - Kohäsion und Kopplung. MSDN Magazin Oktober 2008, Internetseite. Abrufbar unter `http://msdn.microsoft.com/de-de/magazine/cc947917.aspx`; zuletzt besucht am 2. September 2009.

[51] M Squared Technologies. Effective Lines of Code eLOC Metrics for popular Open Source Software. Internetseite. Abrufbar unter `http://msquaredtechnologies.com/m2rsm/rsm_software_project_metrics.htm`; zuletzt besucht am 2. September 2009.

[52] Klein, Elphinstone, Heiser, Andronick, Cock, Derrin, Elkaduwe, Engelhardt, Norrish, Kolanski, Sewell, Tuch, Winwood. seL4: Formal Verification of an OS Kernel. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP)*, Big Sky, MT, Oktober 2009.

[53] Laura L. Pallum. *Software Fault Tolerance - Techniques and Implementation*. Artech House, 2001. ISBN: 1-58053-137-7.

[54] Arlat, Fabre, Rodriguez. Dependability of COTS microkernel-based systems. *IEEE Transactions on Computers*, 51(2):138–163, Februar 2002.

[55] Potyra, Sieh, Cin. Evaluating fault-tolerant system designs using FAUmachine. In *Proceedings of the 2nd International Workshop on Engineering fault tolerant systems (EFTS)*, 2007.

[56] Koopman, Sung, Dingman, Siewiorek, Marz. Comparing Operating Systems Using Robustness Benchmarks. In *Proceedings of the 16th Symposium on Reliable Distributed Systems (SRDS)*, 1997.

[57] Florian Pose. IgH EtherCAT Master 1.5 - Preliminary Documentation, Revision 1828. Technischer Bericht, Ingenieurgemeinschaft IgH, 2009. Abrufbar unter `http://etherlab.org/download/ethercat/ethercat-1.5-r1828.pdf`; zuletzt besucht am 18. September 2009.

[58] Roland Aigner. DICE Version 3.3.0 User's Manual. Technischer Bericht, TU-Dresden, 2007. Abrufbar unter `http://os.inf.tu-dresden.de/dice/`; zuletzt besucht am 23. September 2009.

[59] Free Software Foundation. The GNU Scientific Library. Internetseite. Abrufbar unter `http://www.gnu.org/software/gsl/`; zuletzt besucht am 2. Oktober 2009.

[60] Ingo Molnar. PREEMPT_RT Kernel-Patches. Internetseite. Abrufbar unter `http://www.kernel.org/pub/linux/kernel/projects/rt/`; zuletzt besucht am 25. August 2009.

[61] Betriebssysteme-Gruppe der TU-Dresden. Öffentliches DROPS/TUDOS Repository. Internetseite. Abrufbar unter `http://www.inf.tu-dresden.de/index.php?node_id=1735&ln=en`; zuletzt besucht am 25. August 2009.

[62] Rusty Russell. Hackbench. Internetseite. Abrufbar unter `http://devresources.linux-foundation.org/craiger/hackbench/`; zuletzt besucht am 5. Oktober 2009.

[63] Stefan Manegold. Calibrator. Internetseite. Abrufbar unter `http://monetdb.cwi.nl/Calibrator/`; zuletzt besucht am 27. Oktober 2009.

[64] Thomas Gleixner. Cyclictest. Internetseite. Abrufbar unter `http://rt.wiki.kernel.org/index.php/Cyclictest`; zuletzt besucht am 8. Oktober 2009.

[65] Open Source Automation Development Lab (OSADL). Latency Measurement Box. Internetseite. Abrufbar unter `http://www.osadl.org/Latency-Box.projects-latency-box.0.html`; zuletzt besucht am 25. Oktober 2009.

[66] David A. Wheeler. SLOCcount. Internetseite. Abrufbar unter `http://www.dwheeler.com/sloccount/`; zuletzt besucht am 1. November 2009.

Glossar und Abkürzungsverzeichnis

APIC	Advanced Programmable Interrupt Controller - Nachfolger vom PIC, unterstützt auch die Verteilung von Interrupts in Multiprozessorsystemen
DDE	Device Driver Environment - DDE stellt Schnittstellen und APIs zur Verfügung, welche von Linux-Treibern genutzt werden können, um diese in DROPS wieder zu verwenden
DROPS	Dresden Real-Time Operating System - Ein an der TU-Dresden entwickeltes mikrokernbasiertes Echtzeitbetriebssystem
Fiasco	Fiasco - Echtzeitfähiger Mikrokern, Betriebssystemkern von DROPS
FPGA	Field Programmable Gate Array - umprogrammierbarer digitaler Schaltkreis
HPET	High Precision Event Timer - Timerbaustein mit mehreren unabhängigen Kanälen, kann ohne großen Aufwand auch in Benutzeranwendungen programmiert werden
IPC	Inter Process Communication - im Kontext von L4 ist dies ein Mechanismus zur synchronen Übertragung von Nachrichten zwischen zwei Threads
PIT	Programmable Interrupt Timer - Timerbaustein im Interruptcontroller von x86-basierter Hardware, häufig zur Generierung des Systemtakts verwendet
TLB	Translation Lookaside Buffer - Zwischenspeicher, welchen die MMU nutzt, um die Übersetzung zwischen virtuellen und physischen Adressen zu beschleunigen

TSC	Time Stamp Counter - 64-Bit Register in x86-basierter Hardware, welches linear zum Prozessortakt inkrementiert wird
WCAO	Worst Case Administrative Overhead - beschreibt die maximale Zeitverzögerung, die durch das Betriebssystem oder dessen Services verursacht wird
WCET	Worst Case Execution Time - Maximale Ausführungszeit eines Programms oder eines Programmstücks

Anhang A

A Quellcodestatistik

Die nun folgenden Tabellen zeigen den Umfang des Quellcodes der im Rahmen dieser Arbeit entwickelten Anwendungen. Die Werte der Spalte *SLOC* (*source lines of code*) wurden mit dem Programm *SLOCcount* [66] ermittelt. Die Software unterstützt jedoch nicht das Dateiformat der IDL (*interface definition language*), sodass diese nicht mit in die Berechnung einfließen (durch * gekennzeichnet). Die Werte der Spalte *LOC* (*lines of code*) entsprechen dagegen der Summe aller Quelldateien der einzelnen Komponenten. Falls nicht anders gekennzeichnet, sind alle Angaben Codezeilen der Programmiersprache C.

Komponente	LOC	SLOC
idl	143	*
include	70	46
lib	592	386
server	785	565
Σ	1590	996

Tab. A.1: Quellcode des Master-Moduls

Komponente	LOC	SLOC
idl	19	*
include	23	19
lib	180	130
server	539	352
Σ	761	501

Tab. A.2: Quellcode des Treiber-Moduls

Anwendung	LOC	SLOC
ec_bench	1277	887 (davon 38 Python)
ec_latency	346	249
ec_abstimeout	246	179
ec_rtlws11	384	276
Σ	2253	1561

Tab. A.3: Quellcode der Beispiel-Anwendungen

Technische Universität Ilmenau
Fakultät für Informatik und Automatisierung
Institut für Praktische Informatik und Medieninformatik
Fachgebiet Verteilte Systeme und Betriebssysteme

TECHNISCHE UNIVERSITÄT
ILMENAU

Thesen zur Diplomarbeit

Vergleich von monolithischen- und mikrokernbasierten Betriebssystemarchitekturen und deren Robustheits- und Echtzeiteigenschaften für den Einsatz in industrietauglichen Anwendungen

André Puschmann

1. Mikrokernbasierte Betriebssysteme sind aufgrund der Implementierung ihrer Systemservices in getrennten Adressräumen besonders für Anwendungen interessant, die einen hohen Anspruch an Robustheit und Fehlertoleranz stellen.
2. Auch Echtzeitsysteme profitieren vom Einsatz von Mikrokernen, da diese die Komplexität des Systemkerns verringern. Das Gesamtsystem wird dadurch flexibler und die Vorhersagbarkeit erhöht sich.
3. Erhöhter Kommunikationsaufwand durch getrennte Adressräume führt zu längeren Programmlaufzeiten.
4. Der Aufbau von verlässlichen Computersystemen, die ebenso in der Lage sind, zeitkritische Aufgaben zu bewältigen, ist mit mikrokernbasierten Betriebssystemen möglich.
5. Die zweckmäßige Konfiguration der Prioritäten mehrerer, voneinander abhängiger Threads, stellt in Echtzeitsystemen eine nicht zu unterschätzende Herausforderung dar.