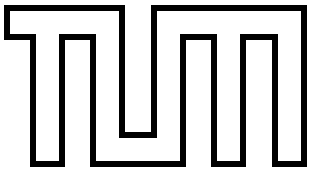


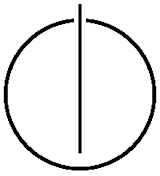
FAKULTÄT FÜR INFORMATIK

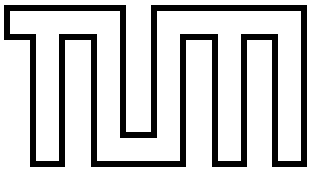
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Bachelorarbeit in Wirtschaftsinformatik

Erweiterung des Fiasco.OC-Mikrokerns um kontextsensitive Schedulingfähigkeiten für sicherheitskritische Anwendungen in eingebetteten Systemen

Valentin Hauner





FAKULTÄT FÜR INFORMATIK

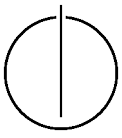
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Bachelorarbeit in Wirtschaftsinformatik

Erweiterung des Fiasco.OC-Mikrokerns um kontextsensitive Schedulingfähigkeiten für sicherheitskritische Anwendungen in eingebetteten Systemen

Extension of the Fiasco.OC microkernel with context-sensitive scheduling abilities for safety-critical applications in embedded systems

Autor:	Valentin Hauner
Aufgabensteller:	Prof. Dr. Uwe Baumgarten
Betreuer:	Daniel Krefft, M.Sc.
Abgabedatum:	15. Oktober 2014



Ich versichere, dass ich diese Bachelorarbeit selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel verwendet habe.

München, den 15. Oktober 2014 Valentin Hauner

Danksagungen

Ein Dank geht an meinen Betreuer Daniel Krefft, der mir bei Recherche, Implementierung und Ausarbeitung wertvolle Unterstützung leistete. Ebenso erwähnt seien die Mitarbeiter am Lehrstuhl für Betriebssysteme der TU Dresden, allen voran Adam Lackorzynski und Björn Döbel, die mir bei der Einarbeitung in den Quellcode behilflich waren. Dabei stellten sie mir freundlicherweise auch weiterführende Informationen zu der Architektur und den Schnittstellen von Fiasco.OC und L4Re zur Verfügung, die über die offizielle Dokumentation hinausgehen.

Kurzfassung

Die vorliegende Arbeit beschäftigt sich mit der Erweiterung des L4-Mikrokerns Fiasco.OC um ein echtzeitfähiges Schedulingverfahren zur Durchführung sicherheitskritischer und kontextsensitiver Aufgaben in eingebetteten Systemen. Um die Integrität und Funktionsfähigkeit dieser Syteme zu gewährleisten, müssen bestimmte Abläufe eine definierte Antwortzeit einhalten und bis zu einem kritischen Zeitpunkt in der Zukunft abgeschlossen sein. Die nachfolgend behandelten kontextsensitiven Abläufe sind dabei aperiodischer Natur, da sie durch ein Ereignis von außen angestoßen werden.

Die Einleitung dieser Arbeit vermittelt die Grundlagen der Mikrokernarchitektur am Beispiel von Fiasco.OC und zeigt typische Problemfelder in eingebetteten Systemen auf, bevor ein Überblick über verschiedene Algorithmen für echtzeitfähiges Scheduling folgt. Schließlich wird dem Leser ein architektonisches Gesamtkonzept zur Integration eines geeigneten Algorithmus in Fiasco.OC präsentiert und dessen konkrete Umsetzung in C und C++ vorgestellt. Um potenzielle Anwendungsgebiete zu veranschaulichen, nehmen die einzelnen Kapitel immer wieder zu den sicherheitsrelevanten Steuerungskomponenten in Automotive Systems Bezug.

Das Ziel dieser Arbeit ist es, dem Leser die Weiterverwendung und Anpassung der Ergebnisse in seinen eigenen Projekten zu ermöglichen. Deshalb wurden sowohl die Konzepte als auch die Schnittstellen so abstrakt und erweiterungsfreundlich wie möglich gestaltet. Alle hinzugekommenen Funktionalitäten sind darüber hinaus umfangreich dokumentiert und mit Beispielen hinterlegt.

Abstract

This work deals with the extension of the Fiasco.OC microkernel with a real-time capable scheduler for safety-critical and context-sensitive applications in embedded systems. To guarantee the integrity and operability of those systems, certain procedures must meet a predefined response time and be finished by a critical point of time in the future. Context-sensitive procedures are triggered by an external event and therefore have an aperiodic character.

The paper's introduction explains the basic concepts of microkernels using the example of Fiasco.OC and reveals typical problems in embedded systems, followed by an overview of different algorithms suitable for real-time scheduling. Finally, the reader gets to know the architecture for integrating a chosen algorithm in Fiasco.OC as well as its concrete implementation in C and C++. Illustrating the potential fields of application, the author repeatedly refers to safety-relevant components in automotive systems.

The work's objective is to enable the reader to use and adapt its results in one's own projects. Therefore, all concepts as well as interfaces are designed as abstract and extensible as possible. Additionally, detailed documentation of the new features with several examples of application is included.

Inhaltsverzeichnis

Gliederung

Teil I: Einführung und theoretische Grundlagen

KAPITEL 1: EINFÜHRUNG

Dieses Kapitel bietet eine kurze Einführung in die theoretischen Grundlagen des Mikrokern-Ansatzes, die Charakteristika von eingebetteten Systemen sowie in die Problemstellungen des Schedulings.

KAPITEL 2: FORSCHUNGSSTAND

Dieses Kapitel vermittelt einen Überblick über den Forschungsstand auf dem Gebiet des echtzeitfähigen Schedulings und fasst die für diese Arbeit relevante Literatur zusammen.

Teil II: Implementierung

KAPITEL 3: CODEBASIS

Dieses Kapitel legt die grundlegende Architektur des Mikrokerns Fiasco.OC und der Laufzeitumgebung L4Re dar und demonstriert die Verwendung der Schnittstellen anhand von Beispielen.

KAPITEL 4: IMPLEMENTIERTER ALGORITHMUS

Dieses Kapitel dokumentiert die vollzogenen Anpassungen in Fiasco.OC und L4Re und erläutert die Funktionalitäten und Einsatzgebiete der neu geschaffenen Bibliothek *libedft*.

KAPITEL 5: ANWENDUNGSBEISPIEL FUR AUTOMOTIVE SYSTEMS

Dieses Kapitel zeigt anhand eines Beispiels für Automotive Systems praktische Anwendungsgebiete der Implementierung für eingebettete Systeme auf.

Teil III: Fazit

KAPITEL 5: ZUSAMMENFASSUNG

Dieses Kapitel fasst wichtige Aspekte der erarbeiteten Problemlösung zusammen und bewertet die Implementierung im Hinblick auf ihre Einsatztauglichkeit in der Praxis.

KAPITEL 6: AUSBLICK

Dieses Kapitel gibt einen Ausblick auf künftige Erweiterungsmöglichkeiten der Implementierung.

Teil I.

Einführung und theoretische Grundlagen

1. Einführung

Das Konzept des Mikrokerns bietet für eingebettete Systeme einen entscheidenden Vorteil: Der Betriebssystemkern stellt nur die elementaren Dienste wie Ressourcenzuteilung und Interprozesskommunikation zur Verfügung und kann daher sehr schlank gestaltet werden. Die eigentliche Basis für die auszuführenden Anwendungen bildet dann die Laufzeitumgebung, die mit dem Mikrokern kommuniziert und für jedes eingebettete System individuell entworfen und konfiguriert wird. Diese Trennung kommt einerseits der teils noch immer vorherrschenden Ressourcenknappheit für solche Systeme zugute, andererseits aber auch dem hohen Anpassungsgrad, dem etwa Steuerungseinheiten in Automobilen unterliegen.

Eine wesentliche Anforderung an viele eingebettete Systeme stellt die Echtzeitfähigkeit dar: Für bestimmte kritische Aufgaben müssen im Voraus verlässliche Aussagen über die Laufzeit unter gegebenen Rahmenbedingungen getroffen werden können. Beispielsweise darf die automatische Bremssteuerung von Automobilen im Gefahrenfall nicht durch andere aktive Prozesse verzögert werden, da dies für die beteiligten Verkehrsteilnehmer im schlimmsten Fall tödlich enden kann. Die verfügbare Prozessorzeit muss also so aufgeteilt werden, dass jede kritische Aktivität zum erforderlichen Zeitpunkt vollständig ausgeführt wird. Angaben über Art und Häufigkeit solcher Tasks sind a priori aber nicht immer bekannt, weshalb verschiedene Abschätzungen mit Worst-Case-Szenarien von Nutzen sein können.

Im Folgenden werden die theoretischen Grundlagen für diese Arbeit eingeführt. Diese umfassen den Mikrokern-Ansatz, Charakteristika von eingebetteten Systemen sowie Verfahren zum Scheduling von Ausführungssträngen.

1.1. Mikrokern-Ansatz

Das Konzept des Mikrokerns beruht darauf, im Systemmodus nur solche Betriebssystemfunktionen auszuführen, die für die Laufzeitumgebung und die im Benutzermodus laufenden Anwendungen unabdingbar sind, wie Tanenbaum in [13] erläutert. Im Gegensatz dazu führen monolithische Kerne erheblich mehr Routinen im privilegierten Modus aus. Der Mach-Kernel, der bereits seit 1985 an der Carnegie Mellon University (CMU) in Pittsburgh entwickelt wurde, war einer der ersten Mikrokern-Implementierungen und Grundstein für viele weitere Entwicklungen [14].

Zu den gängigen Aufgaben eines Mikrokerns gehören die Interprozesskommunikation (IPC), die Speicherverwaltung sowie das Multiplexing von Ausführungssträngen für den Prozessor in Form von Threads. Jedoch zählen anders als bei monolithischen Kernen Dateisysteme, Gerätetreiber und Protokollimplementierungen nicht dazu. Diese werden als so genannte Serverprozesse im Benutzermodus ausgeführt und erhalten Anfragen von Clientprozessen. Abbildung 1.1 veranschaulicht diesen Sachverhalt: Eine Benutzeranwen-

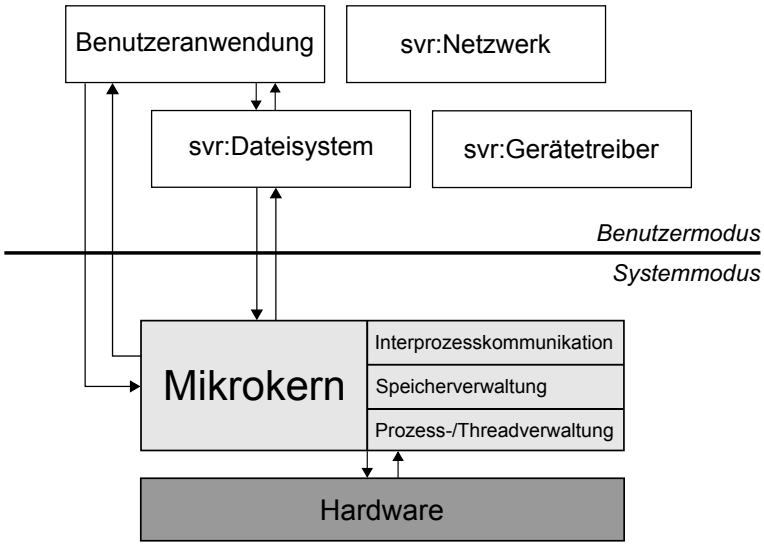


Abbildung 1.1.: Mikrokerne stellen nur Basisdienste bereit, alle darauf aufsetzenden Funktionalitäten laufen im Benutzermodus als Serverprozesse ab.

dung greift sowohl direkt auf die Funktionalitäten des Kerns zu als auch auf die Dienste des Dateisystem-Servers, welcher wiederum eine Anforderung an den Kern sendet.

Ein wesentlicher Vorteil der Mikrokern-Architektur liegt in der hohen Anpassungsfähigkeit des Systems: Da die Serverprozesse vom Betriebssystemkern entkoppelt sind, können sie sehr einfach ausgetauscht werden. So lässt sich etwa das verwendete Dateisystem ohne nennenswerten Aufwand durch ein anderes ersetzen. Einzig die zur Kommunikation eingesetzten Schnittstellen müssen identisch sein. In Systemen mit einem monolithischen Ansatz ist eine derartige Substitution wesentlich schwieriger, da das Dateisystem in der Regel fest im Kern integriert ist.
Infolge der Auslagerung vieler Funktionalitäten in den Benutzermodus wird der Umfang der so genannten Trusted Computer Base (TCB) auf ein Minimum beschränkt. Die TCB ist ein Überbegriff für sämtliche sicherheitsrelevante Artefakte in Hard- und Software; enthält sie Schwachstellen, sind die Sicherheitseigenschaften des gesamten Systems gefährdet. Die geringe Größe der TCB erleichtert es deshalb auch, die Eignung einer Software für sicherheitskritische Zwecke formal zu verifizieren [8].
Ebenso werden die Auswirkungen instabiler Softwarekomponenten auf die Zuverlässigkeit des Gesamtsystems reduziert. Abstürze von Gerätetreibern etwa führen bei einem mikrokernbasierten System aufgrund der Realisierung als Serverprozesse im Benutzermodus in der Regel zu keinem Komplettabsturz. Monolithische Kerne sind anfälliger: Große Teile des Codes bestehen hier aus Treibern, die im privilegierten Modus ausgeführt werden.

Im Allgemeinen weisen mikrokernbasierte Systeme eine schlechtere Performanz als monolithische auf, da der modulare Aufbau mit Client- und Serverprozessen viele Kontextwechsel erfordert [11]. Inzwischen gibt es jedoch einige abgewandelte Mikrokern-Ansätze, die die Prozesskommunikation beschleunigen.

1.1.1. Familie der L4-Mikrokernel

L4 ist die Bezeichnung für eine Familie von Mikrokernen, die inzwischen Implementierungen verschiedener Entwicklerteams umfasst [7]. Der Entwurf geht auf den deutschen Informatiker Liedtke zurück, der in den 90er Jahren die schlechte Performanz der frühen Mikrokern-Implementierungen wie Mach zum Anlass nahm und L4 konzipierte. Liedtke machte die asynchrone Interprozesskommunikation sowie unzureichende Code- und Speicherlokalität als Hauptursache dafür aus und führte die synchrone Nachrichtenkommunikation ein.

1.1.2. L4-Implementierung Fiasco.OC

Der in C++ implementierte L4-Kern Fiasco.OC stammt von der Technischen Universität Dresden und ist Grundlage für deren TUD:OS-Betriebssystem [8]. Das Herzstück von Fiasco.OC ist die beim Start instanziierte *Factory*, die alle weiteren Kernelobjekte wie Tasks, Threads, IPC-Gates sowie Interrupt Requests (IRQ) erzeugt.

Ein Task ist gleichbedeutend mit einem Prozess und stellt einen virtuellen Adressbereich dar. Jeder Thread ist an einen bestimmten Task gebunden und repräsentiert einen eigenen Ausführungsstrang. Threads verfügen einerseits über einen *Kernel-level Thread Control Block (KTCB)* bestehend aus Registern, Instruktionszeiger und Stack sowie andererseits über einen *User-level Thread Control Block (UTCB)*. Letzterer dient als Zwischenspeicher für die Kommunikation mit anderen Threads. Ein IPC-Gate fungiert als Kanal zur Interprozesskommunikation, während IRQ-Objekte zur Steuerung von hard- und softwareseitigen Unterbrechungsanforderungen verwendet werden.

Im Wesentlichen stellt Fiasco.OC seiner Umwelt nur einen einzigen Systemaufruf zur Verfügung: Über die *kinvoke*-Methode können zwei Objekte miteinander kommunizieren.

Kommunikation über Nachrichten

In Fiasco.OC erfolgt die Kommunikation zwischen zwei Threads über den synchronen Austausch von Nachrichten. Der Sender schreibt seine Nachricht dazu in seinen eigenen UTCB und ruft über die *kinvoke*-Methode des Kernels das zum gewünschten Empfänger führende IPC-Gate auf. An dieser Stelle blockiert der Sender, bis die Gegenseite Empfangsbereitschaft signalisiert hat. Schließlich stellt der Kern die Nachricht zu, indem er sie in den UTCB des Empfängers kopiert.

Rechteverwaltung mit Capability-Listen

Der Zugriff auf ein Kernelobjekt durch ein anderes erfolgt in Fiasco.OC nicht durch globale Identifikatoren. Die Entwickler der TU Dresden begründen ihre Entscheidung mit erheblichen Sicherheitsmängeln dieses Konzepts: Clients könnten versuchen, diese Identifikatoren durch Brute-Force-Methodik zu erraten, um sich so unberechtigten Zugriff auf geschützte Objekte zu verschaffen.

Stattdessen erhält jeder Task eine eigene Capability-Liste, die andere Kernelobjekte über einen lokalen Identifikator zugänglich macht. Jedem Task können Capabilities gewährt oder entzogen werden, wodurch eine individuelle Rechteverwaltung ermöglicht wird.

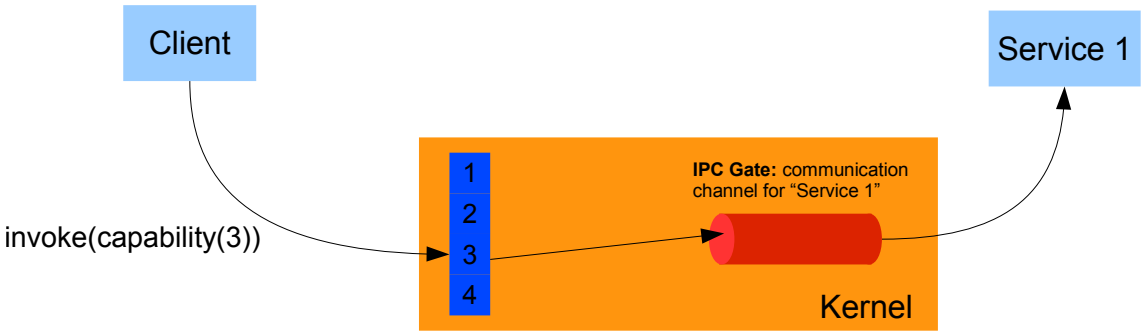


Abbildung 1.2.: Der Client-Thread verfügt über die Capabilities mit Indizes 1 bis 4 und kann mit *Service 1* über die Capability mit Index 3 kommunizieren. Die Capabilities sind nur für diesen Client-Thread gültig. *Bildquelle: [8]*

Entscheidend ist, dass Zugriffe auf Capabilities stets über den Kern erfolgen, denn nur so kann sichergestellt werden, dass der Aufrufer über die erforderlichen Rechte verfügt. Abbildung 1.2 illustriert dieses Verfahren am Beispiel der Kommunikation zwischen einem Client und einem Server.

Resource Mappings mit Flexpages

Threads können ihre Capabilities auch an andere Threads weitergeben. Die für das Mapping notwendige Datenstruktur heißt in Fiasco.OC *Flexpage* und beschreibt, welche Ressource wem und mit welchen Rechten zur Verfügung gestellt wird. Dabei kann unter anderem festgelegt werden, ob der Empfänger die Ressource nur lesen oder gar erneut weitergeben darf. Somit lässt sich ein Adressbereich über mehrere Threads hinweg kaskadierend aufteilen.

1.2. Eingebettete Systeme

Ein eingebettetes System ist ein Computersystem, das für einen spezifischen Einsatzzweck entwickelt wurde und im Gegensatz zu einem Personal Computer (PC) durch den Endanwender nicht frei programmiert werden kann, wie es Heath in [10] formuliert. Prominente Beispiele sind Steuerungseinheiten in Waschautomaten oder Automobilen. Eingebettete Systeme sind in der Regel in größere technische Geräte integriert und als solche nicht erkennbar.

1.2.1. Grundlegende Probleme

Viele eingebettete Systeme unterliegen Echtzeitanforderungen: Hard- und Software müssen in der Lage sein, gewisse Aufgaben zu einem bestimmten Zeitpunkt mit einer definierten Antwortzeit auszuführen. Verpasst ein kritischer Ausführungsstrang seine Deadline, kann dies katastrophale Auswirkungen auf das Gesamtsystem nach sich ziehen. Daher spricht man in diesem Zusammenhang auch von einer harten Echtzeitanforderung.

Verzögert sich hingegen eine Berechnung mit weicher Echtzeitanforderung, ist das Resultat für das System im schlimmsten Fall nutzlos, es entsteht allerdings kein Schaden für die Umwelt. Als Beispiele für harte Echtzeitanforderungen seien die Sensorik in medizinischen Geräten sowie die Bremssteuerung in Automobilen zu nennen; weiche Echtzeitanforderungen treten unter anderem bei der Wiedergabe von Audio- und Videomaterial auf.

Darüber hinaus muss die Hardware von eingebetteten Systemen häufig platzsparend entworfen werden, was in einem hohen Integrationsgrad der verwendeten Bauteile resultiert. Auch die Robustheit gegenüber Erschütterungen, Wärmeentwicklung und sonstigen physikalischen Einflüssen sollte die von herkömmlichen Computern übersteigen, da eingebettete Systeme nicht immer stationär betrieben werden.

Ressourcen wie Strom, Prozessor und Speicher stehen eingebetteten System aufgrund der oben genannten Beschränkungen meist nicht in der gleichen Quantität wie klassischen Computern zur Verfügung, sodass ein sparsamer Umgang mit den Betriebsmitteln sowie eine Feinabstimmung der Software auf die Hardware vonnöten sind.

1.2.2. Eingebettete Systeme in der Automobilindustrie

Automobile verfügen über eine Vielzahl von Sensoren, die Messwerte an verschiedenste Mikrocontroller liefern. Die Kontrolleinheit für das Antiblockiersystem (ABS) zum Beispiel regelt den Bremsdruck jedes einzelnen Reifens anhand der Signale des jeweiligen elektronischen Raddrehzahlsensors. Häufig ist für jede Steuerungseinheit ein separater Controller mit eigenen Schnittstellen verbaut. Da die Nachfrage nach Fahrerassistenzsystemen in der Vergangenheit kräftig angestiegen ist, haben die Hersteller zahlreiche weitere Elektronikkomponenten in die Fahrzeuge integriert. Als nachteilig erweist sich jetzt nicht nur der hohe Installationsaufwand, sondern auch die komplizierte Wartung beim Kunden.

Neue Ansätze sehen vor, die Informationstechnik in Automobilen zu zentralisieren, indem statt einer Vielzahl von Controllern ein leistungsfähiges Steuerungsgerät eingesetzt wird, das die gesamte Kommunikation mit Sensoren und Mechanik übernimmt, wie Bernard et al. in [2] darlegen. Moderne Speicherschutzmechanismen müssen dann dafür Sorge tragen, dass sich die einzelnen Teilsysteme nicht negativ beeinflussen und insbesondere eine Trennung zwischen sicherheitskritischen und unkritischen Funktionalitäten gewahrt bleibt. Hierzu bieten sich schlanke Mikrokerne mit Virtualisierungsfähigkeiten und echtzeitfähigen Schedulingalgorithmen an.

1.3. Scheduling

Scheduling beschreibt die Vergabe von Prozessorzeit an unterschiedliche Ausführungsstränge nach einem definierten Algorithmus. Ausführungsstränge werden meist durch Threads repräsentiert und können sich entweder in Ausführung befinden oder auf die Zuteilung von Rechenzeit warten. Das theoretische Fundament zum Scheduling hat Buttazzo ausführlich in [3] erarbeitet.

Bei dem in dieser Arbeit betrachteten partitionierten Schedulingansatz verfügt jeder Prozessor über eine eigene Ready-Queue mit rechenwilligen Threads. Sobald ein Kontextwechsel ansteht, entnimmt der so genannte Dispatcher den Folgethread aus dieser Queue und teilt ihm CPU-Zeit zu. Der Scheduler entscheidet, wie die Threads in die Queue

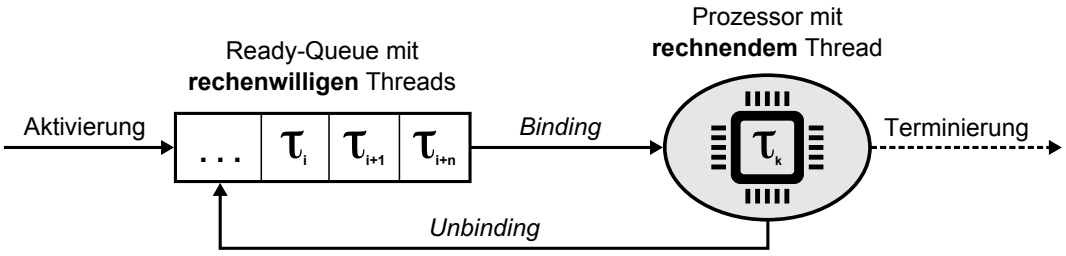


Abbildung 1.3.: Der Scheduler befüllt die Ready-Queue eines Prozessors und entscheidet, welcher Thread als Nächstes zur Ausführung kommt. Das tatsächliche *Binding* und *Unbinding* von Threads an die CPU erledigt der Dispatcher. *Grafik in Anlehnung an [3], Kap. 2, S. 24*

eingereiht werden und welcher Thread als Nächstes zur Ausführung kommt. Abbildung 1.3 stellt den Zusammenhang zwischen Threadzuständen, Ready-Queue, Dispatcher und Scheduler grafisch dar.

Zusätzlich zu den Threadzuständen *rechenwillig* und *rechnend* gibt es noch den Zustand *wartend*, in dem ein Thread verharrt, falls eine vom ihm angeforderte Ressource exklusiv belegt ist. Der Thread wird erst dann wieder als rechenwillig markiert, nachdem die Ressource freigegeben worden ist.

1.3.1. Charakteristika von Schedulingalgorithmen

Schedulingalgorithmen lassen sich anhand zahlreicher Eigenschaften klassifizieren. In den folgenden Abschnitten werden die wichtigsten davon vorgestellt.

Art von Schedulingparametern

Schedulingalgorithmen können auf eine Vielzahl von Threadparametern zurückgreifen, um über die Ausführungsreihenfolge zu entscheiden. Die bekanntesten Vertreter sind prioritätsbasierte Algorithmen, die Threads mit hoher Priorität zuerst rechnen lassen.
In Echtzeitumgebungen ist Ausführungssträngen meist eine Deadline zugewiesen, also ein kritischer Zeitpunkt, zu dem die Ausführung abgeschlossen sein muss. Deadline-Scheduler bemühen sich deshalb, diese Deadlines einzuhalten.
Algorithmen wie Round Robin (RR) weisen Threads ein festes Zeitquantum von wenigen Millisekunden auf dem Prozessor zu und wählen nach Ablauf dieses Slots einen Nachfolgethread aus einer meist nach Aktivierungszeitpunkt sortierten Liste aus.

Veränderlichkeit von Schedulingparametern

Man spricht von dynamischen Schedulingparametern, falls diese während der Laufzeit des Systems an neue Gegebenheiten angepasst werden können. Ein fairer prioritätsbasierter Scheduler zum Beispiel wird die Priorität von rechenintensiven Threads senken, um anderen Threads eine größere Chance zur Ausführung zu geben.
Statische Schedulingparameter hingegen werden lediglich zum Erstellungszeitpunkt initialisiert und bleiben während der gesamten Laufzeit unverändert.

Präemption

Ein Schedulingalgorithmus arbeitet präemptiv, wenn er laufende Threads vor Beendigung ihrer Berechnungen unterbrechen und wieder in die Ready-Queue einreihen darf. Der pausierte Thread kann seine Arbeit erst dann fortsetzen, sobald er den Prozessor erneut zugeteilt bekommt. Die Motivation für diese Unterbrechungen liegt in einem besseren Reaktionsverhalten des Systems: Ausführungsstränge mit hoher Priorität wie das Verarbeiten von Hardwareereignissen sollten möglichst verzögerungsfrei ausgeführt werden, um einen flüssigen Arbeitsablauf zu gewährleisten.
Kommt ein nicht-präemptiver Scheduler zum Einsatz, kann ein rechenintensiver Thread im schlimmsten Fall die Reaktionsfähigkeit des Gesamtsystems zum Erliegen bringen, da der Prozessor keine Möglichkeit hat, andere Aufgaben zu erledigen.

Berechnungszeitpunkt des Schedules

Wird das Schedule, also der Zeitplan für die Ausführung der Threads, vor deren Einreihung in die Ready-Queue berechnet, so handelt es sich um einen Offline-Scheduler. Die erzeugte Abfolge wird dazu in einer Datenbank gespeichert und dann vom Dispatcher nur noch abgearbeitet.
Online-Scheduler hingegen entscheiden über die Abfolge erst zur Systemlaufzeit, etwa bei der Aktivierung und Terminierung von Threads. Aus diesem Grund sind anders als beim Offline-Ansatz a priori keine Informationen über die Ausführungsstränge erforderlich.

1.3.2. Echtzeitfähiges Scheduling

Das Herzstück eines echtzeitfähigen Systems ist ein echtzeitfähiger Schedulingalgorithmus, der sich darum bemüht, alle kritischen Vorgänge innerhalb ihres jeweiligen Antwortfensters abzuschließen. Dieses Ziel lässt sich umso leichter verfolgen, je präziser die Ausführungszeiten der Threads beziffert werden können.

Aperiodische Threads

Im Folgenden wird von einer Menge aperiodischer Threads $T = \{\tau_1, \ldots, \tau_n\}$ ausgegangen, wobei jedem Thread τ_i ein Aktivierungszeitpunkt a_i, eine Ausführungsdauer C_i sowie eine relative Deadline D_i zugewiesen sind. Die absolute Deadline lässt sich leicht durch $d_i = a_i + D_i$ errechnen. Zum Zeitpunkt f_i hat der jeweilige Thread seine Berechnung abgeschlossen. Seine Verspätung ergibt sich somit durch $L_i = f_i - d_i$.

Earliest Due Date Earliest Due Date (EDD) ist ein von Jackson im Jahr 1955 entwickelter Algorithmus, der eine Menge synchron eintreffender Threads als Eingabe erwartet, also $a_i = s \ \forall i \in \{1, \ldots, n\}, s \in N_0$. Präemption ist aufgrund der Synchronität überflüssig.
Der Algorithmus führt die gegebenen Threads aufsteigend nach ihrer Deadline aus und minimiert damit die maximale Verspätung $L_{max} = \max\{L_i, \ldots, L_n\}$. Die Optimalität beweist Buttazzo in [3].

Earliest Deadline First Earliest Deadline First (EDF) geht auf Horn aus dem Jahr 1974 zurück und ist eine Verallgemeinerung von EDD, da die Aktivierungszeitpunkte der einzelnen Threads nun nicht mehr identisch sein müssen. Der Algorithmus führt immer den Thread mit der kleinsten absoluten Deadline aus. Um dies zu erreichen, arbeitet er präemptiv, indem er laufende Threads unterbricht, sobald Threads mit einer kleineren Deadline eintreffen. Auch EDF minimiert die maximale Verspätung L_{max}, der Beweis dazu findet sich wieder in [3].

Periodische Threads

Periodische Threads werden nicht nur einmal, sondern wiederholt in festen Zeitabständen ausgeführt. Jedem periodischen Thread τ_i ist eine Periode T_i zugewiesen, die das Intervall zwischen zwei aufeinanderfolgenden Ausführungen beschreibt. Nachfolgend verfügen alle Instanzen des Threads über die gleiche relative Deadline D_i und die gleiche Ausführungsdauer C_i. Die k-te Instanz $\tau_{i,k}$ von τ_i wird charakterisiert durch ihren Aktivierungszeitpunkt $a_{i,k}$ sowie durch ihre absolute Deadline $d_{i,k}$. Die Ausführung von $\tau_{i,k}$ beginnt bei $s_{i,k}$ und endet bei $f_{i,k}$.

Rate Monotonic Rate Monotonic (RM) weist jedem periodischen Thread τ_i vor dessen Aktivierung eine feste Priorität P_i zu, die während seiner Laufzeit unverändert bleibt. Threads mit einer kleineren Periode, also mit einer höheren Frequenz an Instanzen, erhalten eine höhere Priorität. Darüber hinaus arbeitet RM präemptiv, sodass die Ausführung einer Threadinstanz beim Eintreffen eines Threads mit einer geringeren Periode unterbrochen wird.

Buttazzo zeigt in [3], dass es keinen anderen prioritätsbasierten Algorithmus gibt, der eine Menge an periodischen Threads unter Wahrung der Echtzeitanforderung ausführt, die RM nicht auch zur Ausführung bringen könnte. Entscheidend dafür ist die für jede Instanz konstante Worst-Case-Ausführungsdauer C_i, die kleiner als die relative Deadline D_i sein muss. RM bietet sich daher unter anderem für die Verarbeitung von Sensordaten an, wo im Voraus präzise Angaben über die Periodizität und Ausführungsdauer getroffen werden können.

1.3.3. Resource Sharing

Auf einem System laufen in der Regel sowohl periodische als auch aperiodische Vorgänge ab. Um beide Klassen mit den oben genannten Algorithmen zur Ausführung zu bringen, ist es sinnvoll, mehrere Scheduler parallel zu verwenden.

Zeitgesteuerte, periodische Threads lassen sich mit dem Rate-Monotonic-Verfahren verwalten und unterliegen häufig harten Echtzeitanforderungen. Die Steuerungseinheit eines eingebetteten Systems kann beispielsweise einen periodischen Thread τ_i einsetzen, um die Werte eines Sensors in festen Zeitabständen abzufragen. Für die Antwortzeit des Sensors lässt sich mithilfe physikalischer Kenntnisse eine pessimistische obere Schranke ermitteln, welche als Schätzer für die Ausführungsdauer C_i jeder Threadinstanz dient.

Ereignisgesteuerte und folglich aperiodische Abläufe weisen meist weiche Echtzeitanforderungen auf und werden daher mit Earliest Deadline First oder einem vergleichbaren

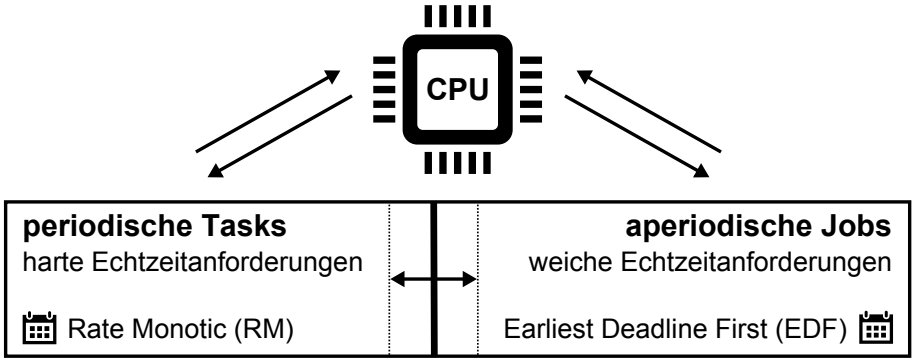


Abbildung 1.4.: Für periodische Tasks mit harten Echtzeitanforderungen sowie für aperiodische Jobs mit weichen Echtzeitanforderungen werden verschiedene Schedulingverfahren eingesetzt.

Algorithmus zur Ausführung gebracht. Als Beispiel sei hier die Steuerung von Multimediakomponenten durch den Fahrer eines Autos zu nennen.

Für den Einsatz unterschiedlicher Schedulingalgorithmen auf einem Rechner sind nicht zwingend mehrere Prozessoren notwendig. Das Konzept des *Resource Sharing* erlaubt es, den beiden Threadarten und somit den eingesetzten Schedulern eine virtuelle Kapazitätsgrenze zuzuweisen, die jederzeit ausgedehnt oder verkleinert werden kann. Abbildung 1.4 verdeutlicht die Idee dahinter. Natürlich skaliert dieses Konzept auch mit zwei und mehr Prozessoren, für deren Koordination Giannopoulou et al. in [9] konkrete Lösungsansätze vorstellen.

2. Forschungsstand

In diese Arbeit flossen sowohl Erkenntnisse aus umfangreichen Standardwerken zu Betriebssystemen als auch Lehrmaterialien und technische Dokumentationen der Technischen Universität Dresden ein.

Zunächst sei *Modern Operating Systems* von Tanenbaum aus dem Jahr 1992 [13] erwähnt, das die Grundlagen für den Entwurf, die Implementierung und den Betrieb von Betriebssystemen darlegt. Tanenbaum arbeitet insbesondere die Unterschiede zwischen monolithischen, mehrschichtigen und mikrokernbasierten Systemen heraus. Sein Werk umfasst darüber hinaus eine detaillierte Einführung in die Konzepte von Prozessen und Threads. Dabei verliert er nie den praktischen Bezug und gibt unter anderem anschauliche Beispiele für die Verwendung der pthread-Bibliothek in C. Der Einblick in die Herausforderungen des Schedulings legt den Grundstein für das zweite verwendete Standardwerk, *Hard Real-Time Computing Systems* von Buttazzo aus dem Jahr 2011 [3].

Buttazzos Konzepte zu echtzeitfähigen Schedulingalgorithmen für aperiodische Prozesse einerseits und für periodische andererseits bilden das theoretische Fundament dieser Arbeit und finden in der nachfolgenden Implementierung Verwendung. Dabei wird sowohl zu seinen Definitionen und Metriken als auch zu seinen Beweisen zur Optimalität und Durchführbarkeit von Schedulingalgorithmen Bezug genommen. Buttazzo gibt außerdem einen Überblick über verschiedene Arten von Schedulingbeschränkungen, wovon für diese Arbeit insbesondere die *Precedence Constraints* relevant sind.

Embedded Systems von Heath aus dem Jahr 2002 [10] liefert ergänzende Informationen zu den Charakteristika von eingebetteten Systemen, während Härtig et al. von der TU Dresden in *The performance of µ-kernel-based systems* [11] die Performanz mikrokernbasierter Systeme am Beispiel des L4-Mikrokerns beurteilen.

Zu Fiasco.OC und L4Re existiert aktuell keine umfassende gedruckte Literatur. Allerdings wird das Projekt regelmäßig in Lehrveranstaltungen der TU Dresden thematisiert, die zugehörigen Lehrmaterialien sind kostenfrei im Internet verfügbar [1]. Die Entwickler pflegen zudem eine umfangreiche Online-Dokumentation samt Wiki und Beispieldatenbank [4], welche auch einige Ausführungen zur Architektur und Implementierung bereithält.

[1] Überblick über die Lehrveranstaltungen der TU Dresden zum Thema Betriebssysteme: `http://www.inf.tu-dresden.de/index.php?node_id=1302`, abgerufen am 1. September 2014.

Teil II.

Implementierung

3. Codebasis

Die Grundlage für die Implementierung bildet der von der TU Dresden entwickelte L4-Kernel Fiasco.OC mit der zugehörigen Laufzeitumgebung L4Re in der Version vom 28. Februar 2014. Der Kernel selbst wird unter der GNU General Public License 2 (GPLv2) veröffentlicht, die Laufzeitumgebung teilweise nur unter der weniger restriktiven GNU Lesser General Public License 2.1.
Zu beiden Komponenten existiert auf der Projektwebsite eine offizielle Dokumentation [4], welche die wichtigsten Schnittstellen erläutert und einige Anwendungsbeispiele bereithält. Interne Funktionen und Methoden sind leider nur vereinzelt dokumentiert.

3.1. Fiasco.OC

Fiasco.OC ist in C, C++ und Assembler verfasst und umfasst knapp 150.000 Zeilen Programmcode. Ein spezieller Präprozessor nimmt dem Entwickler mühsame Arbeit ab: Einerseits automatisiert er die Generierung von Headerdateien, andererseits leistet er wichtige Unterstützung beim Konfigurationsmanagement. Vor der Kompilierung lassen sich mithilfe eines Konfigurationsskripts detaillierte Anpassungen am Kern vornehmen. Der Entwickler platziert dazu im Code so genannte Präprozessortags, die im Konfigurationsskript aktiviert werden können. Auf diese Weise wird festgelegt, welche Funktionalitäten einkompiliert werden sollen.

Zentral für die nachfolgenden Betrachtungen sind die Kernel-Klassen *Task*, *Thread*, *Context*, *Sched_context* und *Ready_queue*. Die Klasse *Thread*, deren Instanzen Ausführungsstränge repräsentieren und jeweils an einen bestimmten Speicherbereich, den so genannten *Task*, gebunden sind, wird von der Klasse *Context* abgeleitet, welche einen Verweis auf eine *Sched_context*-Instanz speichert. Jedem Thread wird hierüber eine Reihe von Schedulingparametern zugeordnet, anhand derer die Ausführungsreihenfolge bestimmt wird. Fiasco.OC verwendet einen partitionierten Schedulingansatz, sodass jeder Prozessor über eine separate Ready-Queue mit rechenwilligen Threads verfügt. Folglich sind alle in einer prozessorspezifischen Queue eingereihten Threads fest an die jeweilige CPU gebunden.

Von großer Bedeutung für das Scheduling sind die Methoden *next_to_run* und *enqueue* der Klasse *Ready_queue*. Erstere gibt einen Verweis auf denjenigen Thread zurück, der als Nächstes zur Ausführung kommen soll, und Letztere fügt den übergebenen Thread an einer geeigneten Position in die Ready-Queue ein. Der Scheduler arbeitet präemptiv und unterbricht daher die laufende Ausführung bei aperiodischen Ereignissen wie Interrupt Requests (IRQ) oder dem Eintreffen neuer Threads. Zudem kommt ein Time-Slice-Verfahren zum Einsatz, wodurch jedem Thread die Rechenzeit nach einem fixen Zeitquantum entzogen wird, um danach eine neue Schedulingentscheidung zu treffen. Die *left*-Methode der Klasse *Sched_context* gibt dabei die verbleibende Dauer im Zeitschlitz an, *replenish* füllt

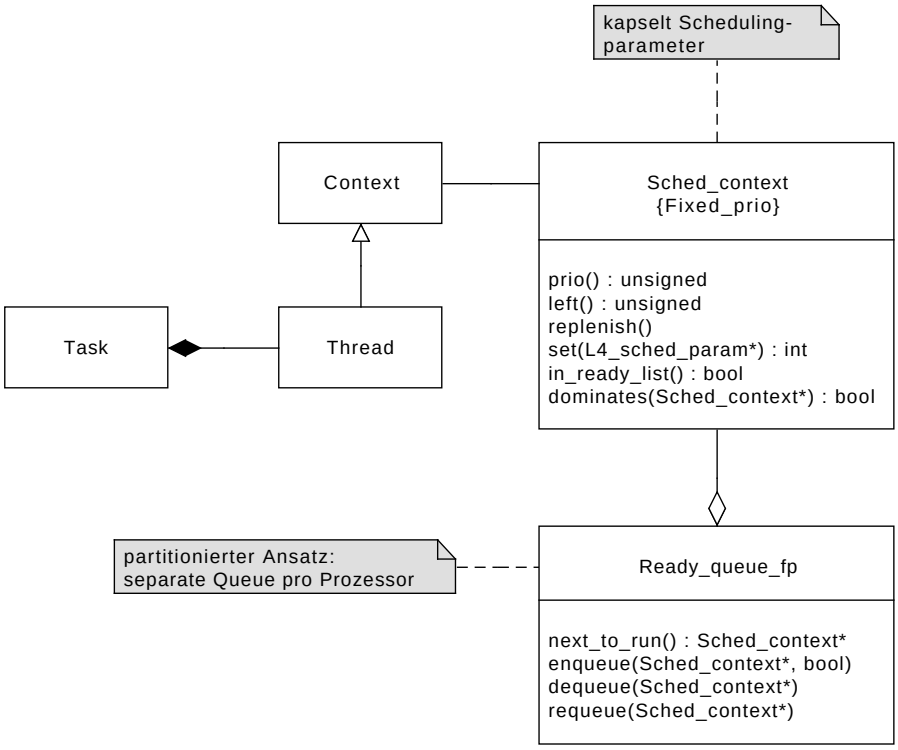


Abbildung 3.1.: Jedem Prozessor wird in Fiasco.OC eine separate Ready-Queue zugewiesen, die rechenwillige Threads samt Schedulingparametern mittelbar speichert. Der Scheduler befüllt diese.

das Quantum nach Ablauf wieder auf.

Das konzeptuelle, stark vereinfachte Klassendiagramm in Abbildung 3.1 veranschaulicht die Zusammenhänge. Da bei Fiasco.OC standardmäßig ein prioritätsbasierter Scheduler zum Einsatz kommt, tritt darin die Klasse *Sched_context* in der Ausprägung *Fixed_prio* auf, die vor der Kompilierung durch die zugehörige Konfigurationsdirektive aktiviert wurde. Hervorzuheben ist, dass es keine Vererbungshierarchie zwischen *Sched_context*-Klassen gibt, sondern der Präprozessor die im Konfigurationsdialog gewählte Ausprägung statisch einsetzt. Verschiedene Ready-Queue-Implementierungen, umgesetzt in getrennten Klassen, können hingegen koexistent zum Einsatz kommen. In der Standardeinstellung verwendet Fiasco.OC für den Prioritätenscheduler die im Diagramm gezeigte *Ready_queue_fp*.

Die Ready-Queue für die 256 unterschiedlichen Prioritätsstufen besteht aus einem ebenso großen Feld aus zyklischen Listen. Jede dieser Listen beinhaltet Threads gleicher Priorität, die nach dem Round-Robin-Verfahren abgearbeitet werden. Die Methode *next_to_run* liefert stets das vorderste Element der höchstprioren Liste, wogegen die Methode *enqueue* neue Threads an das Ende der passenden Prioritätsliste stellt. Die Threadpriorität liest der Algorithmus dabei aus dem zugehörigen *Sched_context*-Objekt aus. *requeue* wird aufgerufen, sobald der Zeitschlitz für einen Thread abgelaufen ist. Zwar fügt die Methode den

Thread nach dem gleichen Verfahren wie *enqueue* ein, setzt den Kopf der Liste aber auf das darauffolgende Element, um das Round-Robin-Verfahren umzusetzen.

Für die Fehlersuche stellt Fiasco.OC einen integrierten Debugger bereit, der zur Laufzeit interaktiv gesteuert werden kann. Die offizielle Bedienungsanleitung [12] erklärt seine wichtigsten Funktionen und Kommandos. Neben Informationen über Kernelzustand und -konfiguration listet der Debugger laufende Threads, deren Backtrace sowie gewünschte Stellen im physischen und virtuellen Speicher auf. Als komfortabel erweist sich sowohl das Überwachen von Seitenfehlern und IPC-Aktionen als auch das Setzen von Haltepunkten. Letzteres kann entweder zur Laufzeit unter Angabe eines Instruktionszeigers oder direkt im Programmcode durch Aufruf des Debuggers erfolgen.

3.2. L4 Runtime Environment

Fiasco.OC allein stellt noch kein lauffähiges Betriebssystem dar. Erst zusammen mit der Laufzeitumgebung *L4 Runtime Environment* (L4Re) lassen sich Anwendungen im Benutzermodus auszuführen. L4Re fungiert dabei als Abstraktionsschicht, die den Anwendungen Basisdienste in Form von Servern bereitstellt und die elementare Kommunikation mit dem Mikrokern übernimmt. Für die Ausführung einer Benutzerapplikation werden neben Fiasco.OC der Root-Task *Moe*, der Root-Pager *Sigma0* und gegebenenfalls der Init-Task *Ned* benötigt, die allesamt Bestandteil von L4Re sind.

In der Standardkonfiguration ist Moe der erste Task, der gestartet wird. Seine Hauptaufgaben bestehen darin, die Systemumgebung zu laden und Dienste für das Ressourcenmanagement zur Verfügung zu stellen, wozu unter anderem die Factory zur Erzeugung von Kernelobjekten, das *Dataspace*-Modul zur Anforderung von Speicherbereichen, das Namespace-Management, ein so genannter Region Mapper zur Koordination mehrerer Pager sowie ein Log-System gehören. Sigma0 als Root-Pager ist für die Auflösung von Page Faults für Moe zuständig.

Benutzerprogramme können wahlweise über ein von Ned interpretiertes Lua-Skript oder über das C-Interface gestartet werden. Während der Einsatz von Lua-Skripten Plattformunabhängigkeit sowie ein hohes Abstraktionsniveau gewährleistet, ist der Weg über das C-Interface meist deutlich flexibler. Die Listings 3.1 und 3.2 stellen beide Möglichkeiten anhand von Minimalbeispielen gegenüber.

Listing 3.1 zeigt das Starten der Binaries *server* und *client* im Read-Only-Bereich des gebooteten Betriebssystems, die jeweils einen eigenen Einstiegspunkt benötigen, etwa in Form einer Main-Routine. Zu Beginn wird ein neuer IPC-Channel angelegt, welcher der Kommunikation zwischen den beteiligten Entitäten dient. Der Server bekommt dazu als Capability die Serverseite mit Schreibrechten zugewiesen, der Client die Clientseite mit Leserechten.

Die deutlich komplexere Prozedur in Listing 3.2 führt zwei Threads in einem gemeinsamen Speicherbereich aus. Der Main-Thread des Tasks fungiert dabei als *thread1*, während *thread2* neu angelegt wird. Dafür erstellt die zuständige Factory nach der Allokation einer Capability ein neues Thread-Objekt, das im Anschluss alle notwendigen Attribute wie Pager, Exception-Handler, Task, Instruction Pointer und Stack Pointer erhält. Als Instruktionszeiger dient hier die parameterlose Funktion *thread2_func* mit leerem Rückgabetyp. Schließlich legt die Routine einen aus Priorität und Quantum bestehenden

```
require("L4");

-- Channel for the two programs to talk to each other
local ex_channel = L4.default_loader:new_channel();

-- The server program, getting the channel in server mode
L4.default_loader:start({ caps = { chan = ex_channel:svr() } },
                        "rom/server");

-- The client program, getting the channel to be able to talk to the server
L4.default_loader:start({ caps = { chan = ex_channel } },
                        "rom/client");
```

Listing 3.1.: Mithilfe eines Lua-Skripts erfolgt das Starten von Benutzeranwendungen auf einem hohen Abstraktionsniveau. *Quelle: [5]; Beispiel verkürzt*

Satz an Scheduling-Parametern an und übergibt den Thread an den Scheduler des Mikrokerns. Tabelle A.1 aus dem Anhang gibt einen Überblick über die verwendeten Bibliotheksfunktionen.

Bei beiden Varianten kommt das Capability-Konzept zum Tragen: Der Zugriff auf Kernelobjekte erfolgt stets über Capabilities, die zentral beim Kernel angefordert und dort verwaltet werden. Der verwendete Typ *l4_cap_idx_t* zur Identifikation der Capabilities ist dabei als vorzeichenlose Ganzzahl implementiert.

```
l4_cap_idx_t thread1_cap, thread2_cap;
unsigned char thread2_stack[8 << 10];

void thread1_func(void) { /* Communicate with thread2 ... */ };
void thread2_func(void) { /* Communicate with thread1 ... */ };

int main(void)
{
  thread1_cap = l4re_env()->main_thread;
  thread2_cap = l4re_util_cap_alloc();

  l4_factory_create_thread(l4re_env()->factory, thread2_cap);

  l4_thread_control_start();
  l4_thread_control_pager(l4re_env()->rm);
  l4_thread_control_exc_handler(l4re_env()->rm);
  l4_thread_control_bind((l4_utcb_t *)l4re_env()->first_free_utcb,
                         L4RE_THIS_TASK_CAP);
  l4_thread_control_commit(thread2_cap);

  l4_thread_ex_regs(thread2_cap,
                    (l4_umword_t)thread2_func,
                    (l4_umword_t)(thread2_stack + sizeof(thread2_stack)), 0);

  // Set priority of thread2 to 5 and quantum to default
  l4_sched_param_t sp = l4_sched_param(5, 0);
  l4_scheduler_run_thread(l4re_env()->scheduler, thread2_cap, &sp);

  // Run thread1
  thread1_func();
}
```

Listing 3.2.: L4Re stellt ein dokumentiertes C-Interface bereit, mit dem sich Kernelobjekte erstellen und steuern lassen. Die Aufrufe sind komplexer als im Lua-Skript, aber auch deutlich flexibler. *Quelle: [6]; Beispiel verkürzt*

4. Implementierter Algorithmus

Für die Erweiterung von Fiasco.OC um einen kontextsensitiven und echtzeitfähigen Scheduler fiel die Wahl auf Earliest Deadline First (EDF). Periodische Verfahren wie Rate Monotonic eignen sich für das in dieser Arbeit verfolgte Ziel nur eingeschränkt, da kontextsensitive Aktivitäten meist aperiodischer Natur sind. Da die Threads außerdem asynchron eintreffen, ist EDF gegenüber Earliest Due Date vorzuziehen.

Allerdings ersetzt das EDF-Verfahren den integrierten Scheduler nicht, sondern ergänzt ihn lediglich. Inspiration dafür leistete die in Fiasco.OC zur Auswahl stehende Kombination von Fixed Priority und Weighted-Fair-Queuing, einem Verfahren zur Ablaufsteuerung in Netzwerken. Das Fortbestehen des prioritätsbasierten Schedulers ist für Systemtasks wie Moe notwendig, da diese aufgrund ihrer Hintergrundaufgaben auf das Time-Slice-Verfahren angewiesen sind und ihnen davon abgesehen keine sinnvolle Deadline zugewiesen werden könnte.

Die folgenden Listings enthalten den Programmcode teils in vereinfachter Form. Insbesondere sind private Attribute und Methoden sowie Routinen zur Fehlerbehandlung nicht aufgeführt, da sie für das grundlegende Verständnis überflüssig sind und den Blick auf die relevanten Stellen erschweren. Der Datenträger im Anhang beinhaltet den vollständigen Code.

4.1. Modifikationen in Fiasco.OC

Das Klassendiagramm in Abbildung 4.1 hebt die Modifikationen an der ursprünglichen Architektur aus Abbildung 3.1 hervor. Die Klassen Task, Context und Thread mussten nicht angepasst werden; deren Implementierungen sind vom gewählten Schedulingalgorithmus weiterhin unabhängig.

Die Klasse *Sched_context* tritt nun in der Ausprägung *fp_edf* auf, deren Bezeichnung für die Kombination aus Fixed Priority und Earliest Deadline First steht. Auszüge aus der Implementierung sind in Listing 4.1 gegeben. Jede Instanz verfügt über ein Typattribut, das den Wert *Fixed_prio* oder *Deadline* annehmen kann. Je nach gewähltem Typ gibt die *metric*-Methode entweder die Priorität oder die Deadline des zugehörigen Threads zurück. Beides lässt sich über den Konstruktor setzen.

Dieser Ansatz hat gegenüber einer Vererbungshierarchie einen entscheidenden Vorteil: Der Typ eines Threads kann zur Laufzeit ohne Castings oder erneute Instanziierungen gewechselt werden. Hierzu kommt die *set*-Methode zum Einsatz, die später bei der Kommunikation zwischen Fiasco.OC und L4Re von Bedeutung sein wird. Als Parameter erhält diese entweder ein Objekt vom Typ *L4_sched_param_fixed_prio* oder *L4_sched_param_deadline*, die beide von *L4_sched_param* abgeleitet sind. Mithilfe einer Fallunterscheidung belegt sie Typ und Metrik des *Sched_context*-Objekts mit den übergebenen Werten, wobei diese intern platzsparend in einer Union gespeichert werden.

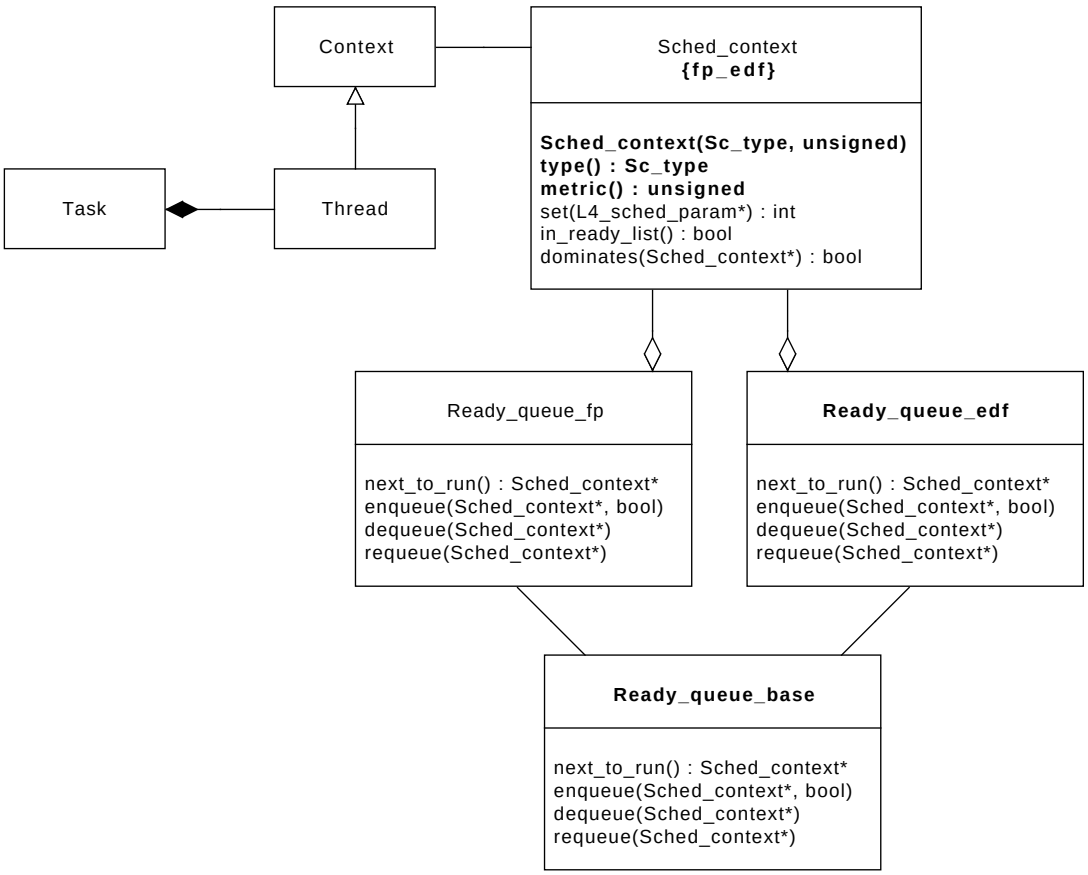


Abbildung 4.1.: Die architektonischen Änderungen belaufen sich auf die Klasse *Sched_context*, die nun in der Ausprägung *fp_edf* auftritt, sowie auf die neuen Klassen *Ready_queue_edf* und *Ready_queue_base*. Erstere setzt den eigentlichen EDF-Algorithmus um, während Letztere den Zugriff auf die beiden koexistenten Ready-Queues koordiniert.

Listing 4.1: Zentrale Methoden der Klasse *Sched_context* in der Ausprägung *fp_edf*

```
struct L4_sched_param_deadline : public L4_sched_param
{
  enum : signed { Class = -3 };
  unsigned deadline;
};

class Sched_context
{
  // Declaring the union Sp needed for the set method ...

public:

  typedef enum Sc_type
  {
    Fixed_prio = 1,
    Deadline   = 2
  }
  Sc_type;

  Sched_context(Sc_type type, unsigned metric)
  {
    if (type == Fixed_prio)
    {
      _t           = Fixed_prio;
      _sc.fp._p    = metric;
      _sc.fp._q    = Config::Default_time_slice;
      _sc.fp._left = Config::Default_time_slice;
    }
    else
    {
      _t           = Deadline;
      _sc.edf._dl  = metric;
      _sc.edf._q   = Config::Default_time_slice;
      _sc.edf._left = Config::Default_time_slice;
    }
  }

  Sc_type type() const
  {
    return _t;
  }

  unsigned metric() const
  {
    if (_t == Fixed_prio)
      return _sc.fp._p;
    else
      return _sc.edf._dl;
  }
```

```
  int set(L4_sched_param const *_p)
  {
    Sp const *p = reinterpret_cast<Sp const *>(_p);

    switch (p->p.sched_class)
    {
      case L4_sched_param_fixed_prio::Class:
        _t          = Fixed_prio;
        _sc.fp._p   = p->fixed_prio.prio;
        if (p->fixed_prio.quantum == 0)
          _sc.fp._q = Config::Default_time_slice;
        else
          _sc.fp._q = p->fixed_prio.quantum;
        break;

      case L4_sched_param_deadline::Class:
        _t          = Deadline;
        _sc.edf._dl = p->deadline.deadline;
        _sc.edf._q  = Config::Default_time_slice;
        break;

      default:
        return L4_err::ERange;
    };

    // Possibly requeue the Sched_context object since its type may have changed
         with this set operation ...

    return 0;
  }
}
```

Neu hinzugekommen ist die Klasse *Ready_queue_edf*, die parallel zur prioritätsbasierten Ready-Queue existiert. Auch hier stehen die Methoden *next_to_run* und *enqueue* im Mittelpunkt, da sie den eigentlichen EDF-Algorithmus umsetzen. *enqueue* reiht die *Sched_context*-Objekte bereits in der für EDF maßgeblichen Reihenfolge in die Queue ein, also aufsteigend nach Deadlines sortiert. *next_to_run* muss folglich immer nur das vorderste Element der Queue an den Aufrufer liefern, was in konstanter Zeit gelingt. Weil häufig entnommen und nur selten eingefügt wird, ergibt sich eine positive Auswirkung auf die Performanz des Schedulingprozesses. Obwohl diese Art des Zugriffs mit den Charakteristika einer klassischen Queue bricht, wird die Datenstruktur im Folgenden dennoch so bezeichnet, um das Benennungsschema beizubehalten.

Wie die Implementierung in Listing 4.2 zeigt, ist die Queue intern als doppelt verlinkte Liste implementiert, wobei *enqueue* mit der Linearen Suche arbeitet, um die richtige Einfügeposition zu ermitteln. Die Methode iteriert über die bestehende Liste und vergleicht die Deadlines der traversierten Objekte mit der des einzufügenden Objekts. Sobald der Algorithmus auf einen Eintrag mit einer größeren Deadline stößt, ist die Suche abgeschlossen und das Element kann an der Position davor eingefügt werden. Wird es ganz vorne eingefügt, muss der Zeiger auf den Listenkopf aktualisiert werden. Objekte mit identischen Deadlines werden gemäß ihrer Eintreffzeitpunkte hintereinander eingereiht. Weiterhin wird sichergestellt, dass kein Objekt mehr als einmal in der Liste enthalten ist.

Listing 4.2: Zentrale Methoden der Klasse *Ready_queue_edf*

```
template<typename E>
class Ready_queue_edf
{
public:

  E * next_to_run() const
  {
    if (!rq.empty())
      return rq.front();

    return idle;
  }

  void enqueue(E *sc_ins, bool)
  {
    if (sc_ins->in_ready_list())
      return;

    if (rq.empty())
      rq.push_front(sc_ins);
    else
    {
      bool inserted = false;
      typename List::Iterator sc_it = List::iter(rq.front());

      do
      {
        if (sc_ins->deadline() < sc_it->deadline())
        {
          rq.insert_before(sc_ins, sc_it);
          inserted = true;
          if (sc_it == List::iter(rq.front()))
            // Inserted at the front of the list, so setting a new head is
                necessary
            rq.rotate_to(sc_ins);
        }
        else if (sc_ins->deadline() == sc_it->deadline())
        {
          rq.insert_after(sc_ins, sc_it);
          inserted = true;
        }
      }
      while (!inserted && ++sc_it != List::iter(rq.front()));

      if (!inserted)
        rq.insert_after(sc_ins, --sc_it);
        // Decrement is necessary as sc_it points to the front of the list after
             having executed the while loop
    }
    count++;
  }
```

```
  void dequeue(E *sc_rm)
  {
    if (!sc_rm->in_ready_list() || sc_rm == idle)
      return;

    rq.remove(sc_rm);
    count--;
  }

  void requeue(E *sc_ins)
  {
    if (!sc_ins->in_ready_list())
      enqueue(sc_ins, false);
  }
}
```

Ebenfalls neu ist die Klasse *Ready_queue_base*: Sie koordiniert den Zugriff auf die beiden koexistenten Ready-Queues und agiert somit als Fassade, indem ihre Methoden abhängig vom Typ des *Sched_context*-Objekts auf der entsprechenden Queue-Instanz operieren. Die Verwendung dieses Entwurfsmusters vereinfacht die Schnittstelle für die zugreifenden Komponenten enorm, da auf diese Weise von den unterschiedlichen Warteschlangen abstrahiert wird. Darüber hinaus ermöglicht diese Form der Architektur die Umsetzung eines flexiblen Resource-Sharing-Konzepts, da beliebige Ready-Queues und ergo Schedulingstrategien ohne Änderung der Schnittstelle hinzugefügt oder entfernt werden können. Die zentrale Positionierung dieser Klasse erlaubt es sogar, mit einer im Benutzermodus angesiedelten *Admission Control* zu kommunizieren, um Zugangs- und Ressourcenbeschränkungen vor dem Einreihen neuer Threads zu prüfen und durchzusetzen.

Aus der Implementierung in Listing 4.3 ist vor allem die Methode *next_to_run* hervorzuheben: Hier werden zuallererst Elemente der prioritätsbasierten Ready-Queue zurückgegeben. Erst wenn diese keine rechenwilligen Threads mehr liefern kann, greift die Methode auf die EDF-Queue zurück. So wird gewährleistet, dass alle aktiven Systemtasks zeitnah zur Ausführung kommen. Dies betrifft unter anderem Moe, aber auch sämtliche Pager-Threads. Jedem Prozessor wird eine separate Instanz dieser Klasse zugewiesen, was dem eingangs erwähnten partitionierten Ansatz entspricht.

Da das Time-Slice-Verfahren tief in Fiasco.OC integriert ist und für prioritätsbasierte Threads weiterhin benötigt wird, wurde es bewusst beibehalten. Dieses gänzlich aus Fiasco.OC zu entfernen, hätte nicht nur umfangreiche Anpassungen tief im Inneren des Mikrokerns bedeutet, die den Rahmen dieser Arbeit sprengen würden, sondern auch zu Inkompatibilitäten beim Einsatz der bereits mitgelieferten Schedulingverfahren geführt. Deshalb belegen sowohl der Konstruktor als auch die *set*-Methode der *Sched_context*-Klasse das Quantumsattribut auch für Deadline-basierte Threads mit Standardwerten. Die Implementierung von *Ready_queue_edf* ignoriert das Ereignis für den Ablauf eines Zeitschlitzes einfach, weil der Round-Robin-Mechanismus für EDF unerwünscht ist: So reiht die beim Ablauf eines Zeitschlitzes aufgerufene *requeue*-Methode den Thread wieder an der gleichen Position in die Queue ein wie zuvor, wogegen die Implementierung der mitgelieferten Fixed-Priority-Queue die Liste entsprechend rotiert.

Listing 4.3: Zentrale Methoden der Klasse *Ready_queue_base*

```
class Ready_queue_base
{
public:

  Ready_queue_fp<Sched_context>  fp_rq;
  Ready_queue_edf<Sched_context> edf_rq;

  Sched_context * next_to_run() const
  {
    Sched_context *s = fp_rq.next_to_run();
    if (s)
      return s;

    return edf_rq.next_to_run();
  }

  void enqueue(Sched_context *sc, bool is_current)
  {
    if (sc->_t == Sched_context::Fixed_prio)
      fp_rq.enqueue(sc, is_current);
    else
      edf_rq.enqueue(sc, is_current);
  }

  void dequeue(Sched_context *sc)
  {
    if (sc->_t == Sched_context::Fixed_prio)
      fp_rq.dequeue(sc);
    else
      edf_rq.dequeue(sc);
  }

  void requeue(Sched_context *sc)
  {
    if (sc->_t == Sched_context::Fixed_prio)
      fp_rq.requeue(sc);
    else
      edf_rq.requeue(sc);
  }
}
```

4.2. Modifikationen in L4Re

Das Interface der Laufzeitumgebung musste dahingehend angepasst werden, dass neben prioritätsbasierten Threads auch solche mit Deadline unterstützt werden. Benutzeranwendungen können die neuen Funktionalitäten nun verwenden, um beide Arten von Threads zu erstellen und an den Mikrokern zu übergeben. Auf die Erweiterung der Lua-Bestandteile wurde bewusst verzichtet, da damit die insbesondere für die Precedence Relations notwendige Flexibilität nicht gewährleistet werden könnte.

Listing 4.4 zeigt die ergänzten und modifizierten Stellen im Code der Bibliothek

l4sys, die für die elementare Kommunikation mit Fiasco.OC zuständig ist und eine Reihe wichtiger Typdefinitionen bereitstellt. Mithilfe des neuen Enumerationstyps *l4_sched_param_type_t* kann der Aufrufer nun selbst festlegen, welche Art von Thread er anlegen möchte, wobei *Fixed_prio* und *Deadline* zur Auswahl stehen. Die Struktur *l4_sched_param_t* erhält zusätzlich zur Priorität ein Deadline-Attribut. Das Quantumsattribut bleibt aus bereits dargelegten Gründen bestehen; Gleiches gilt für den Eintrag *affinity*, der die Bindung des Threads an eine bestimmte CPU beschreibt.

Die neue Funktion *l4_sched_param_by_type*, die als Erweiterung zur bereits bestehenden Funktion *l4_sched_param* entstanden ist, ermöglicht die Instanziierung eines neuen Parameterobjekts. Dazu werden die Art des Threads, die zugehörige Metrik sowie die Länge eines Zeitschlitzes übergeben. Abhängig von der Threadart steht die Metrik entweder für die Priorität oder die Deadline des Threads. Die ursprüngliche Funktion erlaubte diese Differenzierung nicht.

Listing 4.4: Neuerungen in der Bibliothek *l4sys* der L4Re

```
typedef enum l4_sched_param_type_t
{
  Fixed_prio = 1,
  Deadline   = 2
}
l4_sched_param_type_t;

typedef struct l4_sched_param_t
{
  l4_cpu_time_t       quantum;
  unsigned            prio;
  unsigned            deadline;
  l4_sched_cpu_set_t  affinity;
}
l4_sched_param_t;

l4_sched_param_t l4_sched_param_by_type(l4_sched_param_type_t type,
                                        unsigned metric,
                                        l4_cpu_time_t quantum)
{
  l4_sched_param_t sp;
  if (type == Fixed_prio)
  {
    sp.prio     = metric;
    sp.deadline = 0;
  }
  else
  {
    sp.prio     = 0;
    sp.deadline = metric;
  }
  sp.quantum    = quantum;
  // Setting affinity ...
  return sp;
}
```

4.3. Modifikationen an der Schnittstelle zwischen Fiasco.OC und L4Re

Die tatsächliche Kommunikation mit Fiasco.OC findet in der Funktion *l4_scheduler_run_thread* der L4Re-Bibliothek *l4sys* statt. Die Entwickler der TU Dresden haben hier einen etwas irreführenden Namen gewählt, da der Thread nach der Übergabe an den Kern nicht unmittelbar ausgeführt, sondern lediglich in die entsprechende Ready-Queue eingereiht wird. Aus Gründen der Abwärtskompatibilität wurde die Benennung aber beibehalten.

Wie bereits aus der Einführung bekannt ist, geschieht die Kommunikation zwischen zwei Entitäten durch den synchronen Austausch von Nachrichten. Dazu werden die zu sendenden Daten in die Nachrichtenregister des User-level Thread Control Block (UTCB) des Absenders geschrieben und mit einem abschließenden Aufruf von *l4_ipc_call* in den UTCB des gewählten Fiasco.OC-Schedulers kopiert. Dieser liest die Informationen schließlich aus, interpretiert sie und vollzieht das Einfügen des Threads in die passende Queue.

Hinzugekommen ist ein sechstes Nachrichtenregister, das mit dem Wert des Deadline-Attributs befüllt ist. Alle anderen Register wie der Operationscode im ersten, die Angaben zur CPU-Bindung im zweiten und dritten sowie Priorität und Quantum im vierten respektive fünften Register bleiben unverändert. In den um eine Stelle nach hinten verschobenen letzten beiden Registern wird die Flexpage für das Threadobjekt übermittelt. Wichtig für den Empfänger ist vor allem das so genannte Tag der Nachricht. Dessen erster Parameter ist ein benutzerdefiniertes Label, das Aufschluss über die Art der Nachricht geben soll. Der zweite Parameter gibt an, wie viele Register mit Nutzdaten befüllt sind, während der dritte die Anzahl der übergebenen Flexpages enthält. Für Listing 4.5 ergeben sich das Label *L4_PROTO_SCHEDULER*, das eine Scheduler-Operation signalisiert, sowie die Literale 6 und 1. Die einzige hier übergebene Flexpage ist das Threadobjekt selbst.

Listing 4.5: Neuerungen in der Bibliothek *l4sys* der L4Re

```
enum L4_scheduler_ops
{
  L4_SCHEDULER_RUN_THREAD_OP = 1UL
  // ...
};

l4_msgtag_t l4_scheduler_run_thread(l4_cap_idx_t scheduler,
                                    l4_cap_idx_t thread,
                                    l4_sched_param_t const *sp)
{
  l4_utcb_t *utcb  = l4_utcb();
  l4_msg_regs_t *m = l4_utcb_mr_u(utcb);

  m->mr[0] = L4_SCHEDULER_RUN_THREAD_OP;
  // Passing affinity data in m->mr[1] and m->mr[2] ...
  m->mr[3] = sp->prio;
  m->mr[4] = sp->quantum;
  m->mr[5] = sp->deadline;

  m->mr[6] = l4_map_obj_control(0, 0);
  m->mr[7] = l4_obj_fpage(thread, 0, L4_FPAGE_RWX).raw;
```

```
  return l4_ipc_call(scheduler, utcb,
                     l4_msgtag(L4_PROTO_SCHEDULER, 6, 1, 0), L4_IPC_NEVER);
}
```

Damit Fiasco.OC die modifizierte Struktur der Nachricht korrekt interpretiert kann, sind Anpassungen in seiner Scheduler-Klasse erforderlich. Der IPC-Aufruf der L4-Laufzeitumgebung führt zunächst zum Aufruf der zu Beginn erwähnten Funktion *kinvoke*, die konzeptuell die einzige Funktion darstellt, die Fiasco.OC seiner Umwelt zur Verfügung stellt. Diese interpretiert den Operationscode im ersten Nachrichtenregister und leitet die Anfrage an die zuständige Methode der Klasse weiter. Im hier beschriebenen Fall handelt es sich um die Operation *Run_thread* mit der zugehörigen Methode *sys_run*. Natürlich müssen den verfügbaren Scheduler-Operationen im Kernelcode die gleichen Ganzzahlen zugewiesen sein wie im Code der Laufzeitumgebung.

Listing 4.6 zeigt nur die neu hinzugekommenen Codestellen in *sys_run*. Die umfangreichen Abläufe für den Einfügevorgang, das Unterbrechen des laufenden Threads sowie das Erzwingen einer erneuten Schedulingentscheidung sind aus Gründen der Übersichtlichkeit nicht enthalten und finden sich verstreut in den Methoden *migrate*, *do_migration*, *start_migration* und *set_sched_params* der Thread-Klasse sowie in der Methode *schedule* der Context-Klasse des Mikrokerns.

Sobald die Methode *sys_run* festgestellt hat, dass im sechsten Nachrichtenregister eine positive Ganzzahl übermittelt worden ist, ändert sie den Typ des zum Thread gehörenden *Sched_context*-Objekts zu *Deadline*. Dazu legt sie einen neuen Deadline-Parametersatz an und übergibt diesen der in Listing 4.1 vorgestellten *set*-Methode von *Sched_context*, welche die tatsächliche Änderung vollzieht.

Listing 4.6: Erweiterung der Methode *sys_run* der Klasse *Scheduler* in Fiasco.OC

```
class Scheduler
{
public:

  enum Operation
  {
    Run_thread = 1
    // ...
  };

  L4_msg_tag kinvoke(L4_obj_ref ref, L4_fpage::Rights rights, Syscall_frame *f,
                     Utcb const *iutcb, Utcb *outcb)
  {
    // Omitted: Cancelling when receiving no scheduler operation

    switch (iutcb->values[0])
    {
      case Info:        return sys_info(rights, f, iutcb, outcb);
      case Run_thread:  return sys_run(rights, f, iutcb);
      case Idle_time:   return sys_idle_time(rights, f, outcb);
      default:          return commit_result(-L4_err::ENosys);
    }
  }
```

```
private:

  L4_msg_tag sys_run(L4_fpage::Rights, Syscall_frame *f, Utcb const *utcb)
  {
    // ...

    if (utcb->values[5] > 0)
    {
      L4_sched_param_deadline sched_p;
      sched_p.sched_class      = -3;
      sched_p.deadline         = utcb->values[5];
      thread->sched_context()->set(static_cast<L4_sched_param*>(&sched_p));
    }

    // ...
  }
}
```

4.4. libedft: Bibliothek zur Verwaltung von EDF-Threads

Um die neuen Funktionalitäten für echtzeitfähiges Scheduling in einer Benutzeranwendung komfortabel einsetzen zu können, wurde die Bibliothek *libedft* entwickelt, die die Erstellung und Steuerung von EDF-Threads vereinfacht. Sie nimmt dem Nutzer einen Großteil der Kommunikation mit Fiasco.OC ab und bereichert L4Re um so genannte *Precedence Constraints*. libedft wurde in C verfasst, kann aber mit geringem Aufwand auch in anderen Programmiersprachen eingesetzt werden, so zum Beispiel in C++ unter Verwendung der Speicherklasse *extern*.

Die Veröffentlichung unter der GNU General Public License 3 (GPLv3) erlaubt die Weitergabe modifizierter Fassungen der Bibliothek unter Einhaltung des Copyleft-Prinzips. Im Gegensatz zu ihrer Vorgängerversion untersagt die GPLv3 jedoch die so genannte Tivoisierung, also die technisch erzwungene Einschränkung, in einem eingebetteten System nur die vom Hersteller signierte Software einsetzen zu können.

4.4.1. EDF ohne Precedence Constraints

Zunächst werden die grundlegenden Datenstrukturen und Funktionen der libedft-Schnittstelle vorgestellt, wobei Precedence Constraints noch nicht thematisiert werden. Alle nachfolgenden Artefakte stammen aus der Headerdatei *l4/libedft/edft.h*, die für die Nutzung der libedft eingebunden werden muss.

Für die entscheidende Abstraktion von der L4-Laufzeitumgebung sorgt die neu geschaffene Struktur *Edf_thread*, die in Listing 4.7 aufgeführt ist. Instanzen davon werden nachfolgend stets als Threadobjekt bezeichnet.

Die Datenstruktur kapselt alle notwendigen Attribute für einen EDF-Thread. Dazu zählen die so genannte *Worst case execution time* (WCET), also die geschätzte maximale Ausführungsdauer, die Deadline, die Threadfunktion mit optionalem Parameter sowie die zugehörigen L4-Capabilities für den L4-Thread und dessen Task. WCET und Dead-

line sind dabei als vorzeichenlose Ganzzahlen implementiert, die Threadfunktion als typenloser Instruktionszeiger. Zusätzlich ist jedem Thread zur Identifikation und besseren Lesbarkeit eine eindeutige Nummer sowie ein Klartextname zugewiesen. Das Attribut für den Nachfolger im *Precedence Graph* wird erst im nächsten Kapitel eingeführt.

Listing 4.7: Struktur für Threadobjekt

```
typedef struct Edf_thread
{
  unsigned      no;          // Unique number
  char          name[20];    // Name
  unsigned      wcet;        // Worst case execution time
  unsigned      dl;          // Deadline
  void         *func;        // EIP (aka: thread function)
  l4_umword_t   arg;         // Argument to pass to the thread function
  Edf_suc      *suc;         // Successor(s) in precedence graph
  l4_cap_idx_t thread_cap;   // L4 thread capability
  l4_cap_idx_t task_cap;     // L4 task capability
} Edf_thread;
```

Nachdem der Benutzer eine Instanz von *Edf_thread* erstellt hat, kann er auf die Funktionen der libedft zurückgreifen, um mit geringem Aufwand umfangreiche Instanziierungen und Prozeduren in der L4-Laufzeitumgebung anzustoßen und die Kommunikation mit Fiasco.OC zu vollziehen. Während diese Abläufe normalerweise manuell implementiert werden müssten, genügen mit der libedft einige wenige Aufrufe. Tabelle 4.1 gibt einen Überblick über die wichtigsten Funktionen.

Tabelle 4.1.: Schnittstelle der L4Re-Bibliothek *libedft*

Basisfunktionen	

`l4_cap_idx_t edft_create_l4_task(`
`  unsigned map_code,`
`  unsigned map_data);`
	Erstellt einen L4-Task und überträgt ihm grundlegende Capabilities.

`int edft_create_l4_thread(`
`  Edf_thread *thread,`
`  unsigned run);`
	Erstellt einen L4-Thread aus dem übergebenen Threadobjekt.

`int edft_run_l4_thread(` `  unsigned no);`	Übergibt den erstellten L4-Thread an den Mikrokern zur Einreihung in die Ready-Queue.

`int edft_run_l4_threads(void);`	Übergibt alle erstellten L4-Threads an den Mikrokern zur Einreihung in die Ready-Queue.
`void edft_exit_thread(void);`	Beendet die Ausführung des aufrufenden Threads.
`Edf_thread * edft_obj(unsigned no);`	Gibt das Threadobjekt zur übergebenen Threadnummer zurück.

Die Funktion *edft_create_l4_task* legt einen neuen L4-Task an und weist ihm eine Reihe grundlegender Capabilities zu, etwa zur Behandlung von Page Faults, zur Bildschirmausgabe sowie zur Erzeugung weiterer Kernelobjekte. Die beiden Flags legen fest, ob dem neuen Task das Code- und Datensegment des aktuellen Tasks zur Verfügung stehen sollen.

edft_create_l4_thread erhält als ersten Parameter ein Threadobjekt und erstellt daraus einen L4-Thread. Die internen Abläufe ähneln denen aus dem Beispiel für das C-Interface von L4Re in Listing 3.2. Das Flag *run* steuert, ob der erstellte L4-Thread unmittelbar an Fiasco.OC zur Einreihung in die EDF-Queue übergeben werden soll oder nicht. Ist es nicht gesetzt, wird der Thread lediglich im Speicher angelegt; die Übergabe kann dann zu einem späteren Zeitpunkt mit *edft_run_l4_thread* erfolgen oder auch manuell durchgeführt werden. *edft_run_l4_threads* übergibt alle bis zum Aufrufzeitpunkt erstellten L4-Threads an den Mikrokern.

edft_exit_thread beendet die Ausführung des aktuellen Threads und sollte am Ende jeder Threadfunktion aufgerufen werden. Da alle neu erstellten Threads direkt vom Mikrokern ausgeführt werden, ist die Rücksprungadresse auf dem Threadstack nicht gesetzt. Aus diesem Grund inaktiviert die Exit-Funktion den Thread dauerhaft.

libedft legt sämtliche Threadobjekte, die über *edft_create_l4_thread* registriert wurden, in einem internen Threadarray ab, sodass der Benutzer mit der Funktion *edft_obj* und der zugewiesenen Threadnummer jederzeit auf das zugehörige Threadobjekt zugreifen kann.

Das Minimalbeispiel in Listing 4.8 erstellt zunächst einen neuen L4-Task mit Zugriff auf Code- und Datensegment des aktuellen Tasks und initialisiert im Anschluss ein Threadobjekt. Die Reihenfolge für die Initialisierung der Attribute ist durch die Strukturdeklaration festgelegt. Hervorzuheben ist, dass das Attribut für die Thread-Capability noch nicht gesetzt werden kann, da diese erst bei der tatsächlichen Erstellung des Threads alloziert wird, wogegen die Task-Capability bereits bekannt ist und übergeben wird. Soll der Thread in keinem neuen, sondern im aktuellen Task ausgeführt werden, kann als Standardwert die Null übergeben werden. *edft_create_l4_thread* führt schließlich die notwendige Kommunikation mit der L4-Laufzeitumgebung durch und übergibt den Thread aufgrund des gesetzten *run*-Flags unverzüglich an den Kernel.

Listing 4.8: Minimalbeispiel für die Verwendung der libedft

```
#include <l4/libedft/edft.h>
#include <stdio.h>

void new_thread_func(l4_umword_t data)
{
  printf("[new_thread] My deadline is: %u, data passed is: 0x%lx\n",
         edft_obj(0)->dl, data);
  edft_exit_thread();
}

int main(void)
{
  l4_cap_idx_t new_task_cap = edft_create_l4_task(1, 1);
  l4_umword_t data = 0xaffe14;
  Edf_thread new_thread = { 0, "new_thread", 5, 10,
                            new_thread_func, data, NULL, 0, new_task_cap };
  edft_create_l4_thread(&new_thread, 1);
}
```

4.4.2. EDF mit Precedence Constraints

Precedence Constraints, zu Deutsch etwa Vorrangsbeschränkungen, spielen überall dort eine Rolle, wo Threads für ihre Berechnungen auf die Ergebnisse anderer Threads angewiesen sind. Für die Umsetzung eines Augmented-Reality-Systems zum Beispiel, das in die Windschutzscheibe eines Automobils integriert ist, erhält ein Thread von einem Kameramodul Bild- und Positionsdaten, die er an zwei weitere Threads zur simultanen Verarbeitung weitergibt: Der erste Thread übermittelt die Positionsdaten zum Herstellerserver, um Navigationsanweisungen abzurufen, während der zweite Thread die Bilddaten rastert und aufbereitet. Der letzte Thread in der Kette schließlich ist auf die Ergebnisse der beiden vorherigen Ausführungsstränge angewiesen, um alle gewonnenen Informationen zusammenzufügen und dem Fahrer visuell zur Verfügung zu stellen.

Solche Beziehungen werden üblicherweise in einem gerichteten, azyklischen Graphen (DAG) dargestellt. Dieser besteht aus einer Menge von Knoten E, den Threads, sowie aus einer Menge gerichteter Kanten V, jeweils repräsentiert durch ein Tupel (i, j) mit $i, j \in E$. Ein solches Tupel, i.Z. $i \rightarrow j$, sagt aus, dass Thread i vor Thread j ausgeführt werden muss. i ist also unmittelbarer Vorgänger von j. Natürlich gelten auch transitive Abhängigkeiten: Die Notation $i \rightarrow^* j$ steht für einen gerichteten Pfad von i nach j, sodass i mittelbarer Vorgänger von j ist. Abbildung 4.2 zeigt einen derartigen Graphen, der zum obigen Beispiel Bezug nimmt.

Buttazzo klassifiziert in [3] das Bestimmen einer optimalen Ausführungsreihenfolge für Threads mit Precedence Relations im Allgemeinen als NP-schwer. Daher trifft man für das so genannte *Precedence-constrained Sequencing Problem* (PCSP) in der Regel verschiedene vereinfachende Annahmen. In dieser Arbeit wird den Threads im *Precedence Graph* keine Deadline, sondern lediglich eine WCET zugewiesen. Das erleichtert die Sequentialisierung des Graphen aufgrund des höheren Freiheitsgrads deutlich. Die Deadlines werden sodann auf der Basis der erzeugten Sequenz und unter Berücksichtigung der jeweiligen WCET berechnet. Außerdem ist jeder Graph durch genau einen Wurzelknoten definiert,

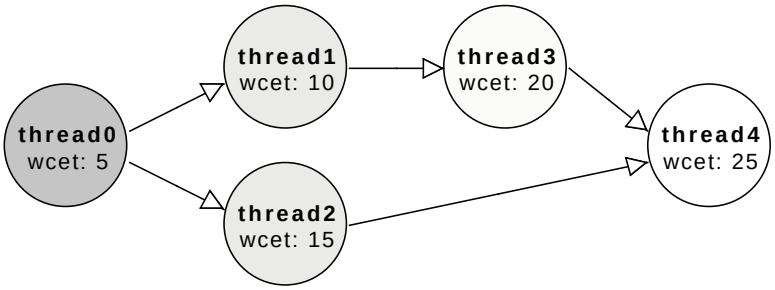


Abbildung 4.2.: Dieser gerichtete azyklische Graph (DAG) zeigt die Precedence Relations zwischen fünf Threads: Bevor *thread1* und *thread2* ausgeführt werden können, muss *thread0* seine Berechnungen abgeschlossen haben. *thread3* wiederum kann erst nach *thread1* starten. *thread4* kommt erst am Schluss an die Reihe.

von dem aus die Traversierung erfolgen kann.

Bereits in Listing 4.7 hat die Struktur für Threadobjekte ein Attribut namens *suc* erhalten, das auf den Nachfolger eines Threads im Graphen verweist. Dessen Datentyp *Edf_suc* implementiert dabei eine verlinkte Liste, wie die Deklaration in Listing 4.9 zeigt. Jedem Threadobjekt können somit beliebig viele Nachfolger zugewiesen werden, die der Sequentialisierungsalgorithmus abarbeitet.

Listing 4.9: Struktur für Threadnachfolger

```
typedef struct Edf_suc
{
  Edf_thread *data;
  Edf_suc    *next;
} Edf_suc;
```

Die Schnittstelle der libedft wird um die beiden Funktionen aus Tabelle 4.2 erweitert: *edft_create_dependent_l4_threads* führt die Sequentialisierung eines Graphen durch und ermittelt dadurch die Ausführungsreihenfolge der enthaltenen Threads. Dabei kommt die Tiefensuche zum Einsatz, die noch nicht besuchte Nachfolgerknoten stets an das Ende der Sequenz stellt und durch wiederholtes Permutieren die Einhaltung der Precedence Relations sicherstellt. Als erster Parameter ist der Wurzelknoten des abzuarbeitenden Graphen zu übergeben, der Wert des zweiten Parameters namens *offset* dient der Verschiebung der Threads auf der Zeitachse und wird auf alle Deadlines aufaddiert. Das Flag *run* legt wie zuvor fest, ob die erstellten L4-Threads sofort an den Mikrokern übergeben werden sollen oder nicht. Die Funktion *edft_print_precedence_graph* erzeugt eine grafische Darstellung des spezifizierten Graphen und schreibt diese auf die Standardausgabe. Um die Ausgabe zu verkürzen, kann anstatt der Wurzel auch nur der Startknoten für den relevanten Teil des Graphen übergeben werden.

Tabelle 4.2.: Erweiterung der libedft um Precedence Constraints

Funktionen für Precedence Constraints	

`void edft_create_dependent_l4_threads(`
`  Edf_thread *root,`
`  unsigned offset,`
`  unsigned run)`
	Errechnet aus den Precedence Relations die Ausführungsreihenfolge und erstellt die zugehörigen L4-Threads.

`void edft_print_precedence_graph(` `  Edf_thread *thread);`	Schreibt eine grafische Repräsentation des *Precedence Graph* beginnend ab dem übergebenen Threadobjekt auf die Standardausgabe.

Listing 4.10 demonstriert die Verwendung der neuen Bibliotheksfunktionen und greift das Beispiel aus Abbildung 4.2 auf. Zunächst werden die fünf Threadobjekte instanziiert, wobei jeweils nur die WCET gesetzt ist; die Deadline bleibt stets unbelegt. Danach werden die Relationen aus dem Graphen in die Listenstruktur übertragen und den Threadobjekten zugewiesen. Der Aufruf von *edft_create_dependent_l4_threads* mit *thread0* als Wurzelknoten ermittelt schließlich die Ausführungsreihenfolge und legt die L4-Threads samt errechneter Deadlines an, die im Anschluss an Fiasco.OC übermittelt werden.

Listing 4.10: Minimalbeispiel für die Verwendung der erweiterten libedft

```
// ...

int main(void)
{
  Edf_thread thread0 = { 0, "thread0",  5, -1, new_thread_func, 0, NULL, 0, 0 };
  // Initializing thread1 to thread4 ...

  // Set up precedence relations
  Edf_suc suc0f = { &thread2, NULL };
  Edf_suc suc0  = { &thread1, &suc0f };

  Edf_suc suc1  = { &thread3, NULL };
  Edf_suc suc2  = { &thread4, NULL };
  Edf_suc suc3  = { &thread4, NULL };

  thread0.suc = &suc0;
  thread1.suc = &suc1;
  thread2.suc = &suc2;
  thread3.suc = &suc3;

  // Create threads according to their precedence relations
  edft_create_dependent_l4_threads(&thread0, 0, 1);
  edft_print_precedence_graph(&thread0);

  return 0;
}
```

5. Anwendungsbeispiel für Automotive Systems

Das nachfolgende Beispiel demonstriert, wie mit Hilfe der libedft sicherheitsrelevante (kritische) und nicht-sicherheitsrelevante (unkritische) Steuerungsvorgänge eines Automotive Systems in getrennten Speicherbereichen ausgeführt werden können. Beide Arten von Operationen sind aperiodischer Natur und können durch ein externes Ereignis ausgelöst werden. Durch die Aufteilung in separate Tasks wird sichergestellt, dass kritische Vorgänge stets isoliert ablaufen und keiner sicherheitsgefährdenden Beeinflussung durch andere Komponenten unterliegen. Dadurch lässt sich die eingangs erwähnte Bündelung der Steuerungseinheiten auf einige wenige Mikrocontroller sicherheitskonform umsetzen.

Der Programmcode in Listing 5.1 wurde stark vereinfacht und um sämtliche Fehlerbehandlungsroutinen verkürzt. Die vollständige Implementierung findet sich auf dem beigefügten Datenträger. Als Testumgebung kam ein mit dem freien Emulator *QEMU* aufgesetztes x86-System zum Einsatz, das sich mithilfe eines speziellen Targets aus dem *make*-Skript der L4-Laufzeitumgebung bequem starten lässt.

Zunächst wird eine neue Datenstruktur deklariert, die eine Operation durch ihren eindeutigen Code und Klartextnamen beschreibt. Instanzen dieser Struktur werden im globalen zweidimensionalen Array *ops* abgelegt. Die erste Dimension gibt dabei an, ob es sich um eine kritische (0) oder unkritische (1) Operation handelt. Die *main*-Funktion alloziert für die beiden Datenbanken *ops[0]* und *ops[1]* jeweils einen eigenen Speicherbereich auf dem Heap und legt zwei Beispielinstanzen pro Typ an: Als kritisch zählt das An- und Abschalten des Allradantriebs, als unkritisch das An- und Abschalten des WLAN-Moduls für das Infotainment-System des Autos.

Für die kritischen und unkritischen Operationen werden zwei separate Tasks erstellt: *task_crit* und *task_uncrit*. *thread0* ist derjenige Thread, der die kritischen Operationen durchführt, und wird deshalb an *task_crit* gebunden. *thread1* ist für die unkritischen Operationen zuständig und wird *task_uncrit* zugewiesen. Der Instruktionszeiger beider Threads wird auf die Funktion *execute_ops* gesetzt, wobei deren Parameter mit dem jeweiligen Index im *ops*-Array belegt wird. *thread0* sollte eine möglichst niedrige Deadline erhalten, da die Verzögerung kritischer Abläufe negative Auswirkungen auf die Funktionsfähigkeit des gesamten Systems haben kann.

Entscheidend ist das Einbinden der jeweiligen Operationsdatenbank in den Speicherbereich des zuständigen Tasks mit der Hilfsprozedur *map_ops_to_task*, die eine nur lesend zugreifbare Flexpage erstellt und dann auf die Funktion *l4_task_map* aus der L4Re-Bibliothek zurückgreift, um das eigentliche Mapping durchzuführen. *ops[0]* wird dabei ausschließlich in *task_crit* eingebunden, während *ops[1]* lediglich für *task_uncrit* sichtbar gemacht wird. Auf diese Weise ist sichergestellt, dass kritische Operationen stets isoliert ausgeführt werden. *thread1* darf keinesfalls im Task der *main*-Routine laufen, da er unter diesen Umständen auf die kritischen Vorgänge zugreifen könnte.

Nach der Übergabe der Threads an den Mikrokern löst die *main*-Funktion beispielhaft die Ausführung einiger Operationen aus. Dazu schreibt sie die Codes der durchzuführenden Operationen in das Nachrichtenregister und startet deren Übermittlung an den jeweiligen Empfängerthread mit einem Aufruf der Routine *l4_ipc_call*, die für die synchrone Interprozesskommunikation zuständig ist. Dieser Datenaustausch kann prinzipiell an einer beliebigen Stelle im System erfolgen, etwa im Steuerungsmodul eines Sensors. *thread0* wird mit der Operation 5 beauftragt, die den Allradantrieb aktiviert. *thread1* soll die Operation 10 ausführen und damit das WLAN-Modul einschalten.

Die Threads warten in der Funktion *execute_ops* nach ihrer Aktivierung mit *l4_ipc_wait* auf eingehende Operationscodes, die sie daraufhin iterativ abarbeiten, und geben dem Aufrufer im Anschluss mit *l4_ipc_reply_and_wait* eine Statusrückmeldung. Dabei können sie stets nur auf den für sie bestimmten Teil des *ops*-Arrays zugreifen. Zu Demonstrationszwecken wird an *thread1* zusätzlich die Operation 6 übermittelt, die als kritisch definiert wurde und den Allradantrieb deaktiviert. Da innerhalb des Tasks von *thread1* aber infolge der getroffenen Speicherschutzmaßnahmen auf die kritischen Operationen in *ops[0]* nicht zugegriffen werden kann, schlägt der Vorgang fehl.

Listing 5.1: Beispiel für die Ausführung kritischer und unkritischer Steuerungsvorgänge eines Automotive Systems in getrennten Speicherbereichen
Quelle: libedft/examples/automotive-example/main.c; Darstellung vereinfacht

```
#include <l4/libedft/edft.h>
// ...

typedef struct Op
{
  unsigned op_code;
  char     op_name[40];
} Op;

Op *ops[2];

void execute_ops(l4_umword_t no)
{
  l4_msgtag_t tag;
  l4_umword_t label;
  l4_msg_regs_t mr;
  unsigned i;

  tag = l4_ipc_wait(l4_utcb(), &label, L4_IPC_NEVER);
  while (1)
  {
    memcpy(&mr, l4_utcb_mr(), sizeof(mr));
    for (i = 0; i < l4_msgtag_words(tag); i++)
    {
      // Executing operation with code mr.mr[i] stored in ops[no] ...
    }

    tag = l4_ipc_reply_and_wait(l4_utcb(), l4_msgtag(0, 0, 0, 0),
                                &label, L4_IPC_NEVER);
  }
}
```

```
void map_ops_to_task(unsigned no, l4_cap_idx_t task_cap)
{
  unsigned fpage_size;
  // Calculating fpage_size ...
  l4_task_map(task_cap, L4RE_THIS_TASK_CAP,
              l4_fpage(ops[no], fpage_size, L4_FPAGE_RO),
              l4_map_control(ops[no], 0, L4_MAP_ITEM_MAP));
}

int main(void)
{
  // ops[0]: Critical operations
  ops[0] = malloc(2 * sizeof(Op));
  ops[0][0] = (Op){ 5, "ENABLE_ALL_WHEEL_DRIVE"  };
  ops[0][1] = (Op){ 6, "DISABLE_ALL_WHEEL_DRIVE" };

  // ops[1]: Uncritical operations
  ops[1] = malloc(2 * sizeof(Op));
  ops[1][0] = (Op){ 10, "ENABLE_WIRELESS_LAN"  };
  ops[1][1] = (Op){ 11, "DISABLE_WIRELESS_LAN" };

  // Separate task for critical operations
  l4_cap_idx_t task_crit_cap = edft_create_l4_task(1, 1);
  Edf_thread thread0 = { 0, "thread0", 2, 5,
                         execute_ops, 0, NULL, 0, task_crit_cap };
  map_ops_to_task(0, task_crit_cap);

  // Separate task for uncritical operations
  l4_cap_idx_t task_uncrit_cap = edft_create_l4_task(1, 1);
  Edf_thread thread1 = { 1, "thread1", 4, 20,
                         execute_ops, 1, NULL, 0, task_uncrit_cap };
  map_ops_to_task(1, task_uncrit_cap);

  edft_create_l4_thread(&thread0, 1);
  edft_create_l4_thread(&thread1, 1);

  l4_msg_regs_t *mr = l4_utcb_mr();

  // Communicate with thread0
  mr->mr[0] = 5;
  l4_ipc_call(thread0.thread_cap, l4_utcb(),
              l4_msgtag(0, 1, 0, 0), L4_IPC_NEVER);

  sleep(5);

  // Communicate with thread1
  mr->mr[0] = 10;
  mr->mr[1] =  6;
  l4_ipc_call(thread1.thread_cap, l4_utcb(),
              l4_msgtag(0, 2, 0, 0), L4_IPC_NEVER);

  return 0;
}
```

Teil III.

Fazit

6. Zusammenfassung

Fiasco.OC eignet sich zusammen mit L4Re hervorragend für die Verwendung in eingebetteten Systemen verschiedenster Art, da beide Komponenten für universelle Einsatzzwecke konzipiert wurden, vielseitig erweiterbar sind und für zahlreiche Zielplattformen zur Verfügung stehen. Die Umsetzung des in dieser Arbeit verfolgten Ziels, den Mikrokern um kontextsensitive Schedulingfähigkeiten für echtzeitfähige und sicherheitskritische Anwendungen zu bereichern, erforderte dennoch eine längere Einarbeitungszeit, um sich mit dessen Architektur und Schnittstellen vertraut zu machen.

Letztlich erstrecken sich die Anpassungen aufgrund des modularen Aufbaus der Software über viele verschiedene Komponenten und Klassen, die zunächst mithilfe einer detaillierten Codeanalyse ausfindig gemacht werden mussten. Das im Hauptteil vorgestellte Klassenmodell legt dabei den Grundstein für die Implementierung und gewährleistet auch künftig die Unabhängigkeit aller für das Scheduling relevanten Algorithmen und Daten von anderen Kernelobjekten wie Tasks und Threads. Gleichzeitig flexibilisiert das verwendete Entwurfsmuster Fassade das Hinzufügen und Entfernen von Schedulingstrategien und ermöglicht die Einbindung einer zentralen Admission Control, um etwaige Sicherheitsrichtlinien beim Eintreffen neuer Ausführungsstränge durchzusetzen.

Earliest Deadline First erfüllt als Schedulingalgorithmus einerseits die Anforderungen an die Echtzeitfähigkeit und bietet sich andererseits wegen der Aperiodizität und Kontextsensitivität der auszuführenden Aufgaben an. Da der prioritätsbasierte Algorithmus Bestandteil des Mikrokerns bleibt, ist die Kompatibilität zu bereits bestehenden Modulen und Anwendungen sichergestellt. Zugleich wird die Grundlage für die Verzahnung der beiden Scheduler gelegt: So können Threads koexistieren, die nach unterschiedlichen Verfahren zur Ausführung gebracht werden, wobei sie ihren Scheduling-Kontext sogar zur Laufzeit wechseln dürfen.

Die Nutzung der Neuerungen kann entweder direkt über das angepasste C-Interface von L4Re oder aber über die eigens entwickelte Bibliothek *libedft* erfolgen. Letztere bietet dem Anwender einen höheren Komfort sowie zusätzliche Funktionalitäten. libedft erlaubt die Erstellung neuer Tasks und EDF-Threads durch den Aufruf einiger weniger Prozeduren und übernimmt dabei die umfangreiche Kommunikation mit der Laufzeitumgebung L4Re, welche die erhaltenen Informationen wiederum an Fiasco.OC weitergibt. Die Aufteilung von Threads in verschiedene Tasks ermöglicht die Durchführung sicherheitskritischer Operationen in separaten Speicherbereichen, welche insbesondere in den in dieser Arbeit thematisierten Automotive Systems auftreten. Darüber hinaus kann der Nutzer Angaben zur Ausführungsreihenfolge der Threads machen, die dann bei der Einreihung in die Ready-Queue des Schedulers berücksichtigt werden. Diese so genannten Precedence Constraints können zur Veranschaulichung auch in einem Graphen auf dem Bildschirm ausgegeben werden.

Obwohl die Schnittstelle der libedft bewusst abstrakt und erweiterungsfreundlich gestaltet wurde, kann die Bibliothek in der vorgestellten Form bereits zu Produktivzwecken

eingesetzt werden. Der vollständig unter der GPLv3 stehende Programmcode ist aber ausdrücklich zur Anpassung an die Erfordernisse und Besonderheiten des jeweiligen Einsatzgebiets gedacht. Für die Verwendung in Automotive Systems etwa ergibt es Sinn, diejenigen Strukturen und Prozeduren, die für die Ausführung kritischer und unkritischer Operationen benötigt werden, stärker mit der Bibliothek zu verknüpfen.

7. Ausblick

Wie bei jedem neuen Softwareartefakt gibt es auch für dieses Projekt viele verschiedene Spielräume zur Erweiterung und Abänderung. So können die zur Speicherung von Deadlines geschaffenen Datenstrukturen mit geringem Aufwand zur Umsetzung anderer Deadline-basierter Schedulingalgorithmen als EDF genutzt werden. Die Prozeduren der libedft müssen dafür nur marginal verändert werden, denn die tatsächliche Reihung der Ausführungsstränge findet ausschließlich in der Ready-Queue-Implementierung von Fiasco.OC statt. Ebenso ist durch die Modifikation einiger Interna des Mikrokerns die Abänderung hin zu einem nicht-präemptiven Verfahren möglich: Die Unterbrechung sicherheitskritischer Operationen zum Beispiel könnte untersagt werden, während unkritischen Aufgaben der Prozessor weiterhin entzogen werden darf.

Erweiterungsmöglichkeiten bestehen darüber hinaus bei den Precedence Constraints: Obwohl das Sequentialisierungsproblem als NP-schwer einzuordnen ist, sobald den Threads die Deadlines a priori zugewiesen werden, gibt es verschiedene approximative Ansätze, um dennoch eine Ausführungsreihenfolge zu erhalten, die nah am Optimum liegt. Einige davon nennt Buttazzo in [3], darunter Latest Deadline First sowie einen Transformationsalgorithmus, der die voneinander abhängigen Threads in eine unabhängige Form überführt, indem deren Ankunftszeiten manipuliert werden. Voraussetzung für beide Verfahren ist, dass aus den gegebenen Relationen überhaupt eine zulässige Reihung für den Scheduler erzeugt werden kann. Wünschenswert wäre hierfür eine Routine zur Überprüfung der Durchführbarkeit, die im Vorhinein ausgeführt wird. Zudem kann es sinnvoll sein, diese Beziehungen ebenso für Threads zu definieren, die auf verschiedenen Prozessoren zur Ausführung gebracht werden.

Auch verschiedene hierarchisch gestaltete Schedulingansätze können vonnutzen sein. Dem von Abeni und Buttazzo in [1] vorgestellten Konzept des *Constant Bandwidth Server* (CBS) liegt ein zweistufiges Verfahren zugrunde: Auf der obersten Ebene arbeitet ein EDF-Scheduler, der darunter liegende Container verwaltet. Jedem dieser Container wird ein festes Budget an Rechenzeit sowie ein oder mehrere aperiodische Jobs mit Deadline zugewiesen. Da immer nur das jeweilige Budget ausgeschöpft werden kann, wirkt sich die Überschreitung einer Deadline durch einen Thread nicht auf Operationen außerhalb des entsprechenden Containers aus. Auf diese Weise wird eine zeitliche Isolation der einzelnen Container sichergestellt, was insbesondere bei der Koexistenz von weichen und harten Echtzeitaufgaben von Vorteil ist.

Obwohl schon jetzt mehrere Schedulingverfahren parallel eingesetzt und dank des Fassadenentwurfs bequem verwaltet werden können, ist das Konzept des Resource Sharing noch nicht vollständig implementiert. Ergänzt werden muss in jedem Fall noch die Möglichkeit, den einzelnen Schedulern virtuelle Kapazitätsgrenzen zuzuweisen, um eine bedarfsgerechte Aufteilung der CPU-Zeit zu garantieren. Ebenso fehlt eine Koordination dieses Skalierungsverfahrens für den Einsatz auf mehreren Prozessoren.

Anhang

A. Dokumentation von L4Re

Tabelle A.1.: Auszug aus der L4Re-Dokumentation [4] mit den für diese Arbeit relevanten Bibliotheksfunktionen der C-Schnittstelle

<table>
<tr><th colspan="2">Initial Environment</th></tr>
<tr><td><code>l4re_env_t * l4re_env(
void)</code></td><td>Gibt einen Zeiger auf die Initialumgebung zurück.</td></tr>
<tr><td colspan="2"><pre>
struct l4re_env_t {
  l4_cap_idx_t parent;
   // Parent object-capability.
  l4_cap_idx_t rm;
   // Region map object-capability.
  l4_cap_idx_t mem_alloc;
   // Memory allocator object-capability.
  l4_cap_idx_t log;
   // Logging object-capability.
  l4_cap_idx_t main_thread;
   // Object-capability of the first user thread.
  l4_cap_idx_t factory;
   // Object-capability of the factory available to the task.
  l4_cap_idx_t scheduler;
   // Object capability for the scheduler set to use.
  l4_cap_idx_t first_free_cap;
   // First capability index available to the application.
  l4_fpage_t utcb_area;
   // UTCB area of the task.
  l4_addr_t first_free_utcb;
   // First UTCB within the UTCB area available to the application.
}
</pre></td></tr>
<tr><th colspan="2">Capability Allocation</th></tr>
<tr><td><code>l4_cap_idx_t l4re_util_cap_alloc(
void)</code></td><td>Liefert einen Index auf eine freie Capability zurück.</td></tr>
<tr><td><code>void l4re_util_cap_free_um(
l4_cap_idx_t cap)</code></td><td>Gibt die übergebene Capability wieder frei (free & unmap).</td></tr>
</table>

Factory	

`l4_msgtag_t l4_factory_create_task(`
`  l4_cap_idx_t  factory,`
`  l4_cap_idx_t  target_cap,`
`  l4_fpage_t const utcb_area)`
	Erstellt einen neuen Task unter der Capability *target_cap* im Speicherbereich *utcb_area*.

`l4_msgtag_t l4_factory_create_thread(`
`  l4_cap_idx_t factory,`
`  l4_cap_idx_t target_cap)`
	Erstellt einen neuen Thread unter der Capability *target_cap*.

Thread	

`l4_msgtag_t l4_thread_ex_regs(`
`  l4_cap_idx_t thread,`
`  l4_addr_t  ip,`
`  l4_addr_t  sp,`
`  l4_umword_t  flags)`
	Setzt Instruction Pointer (IP), Stack Pointer (SP) und bestimmte Flags des Threads. Die Flags ermöglichen die Unterbrechung von laufender Interprozesskommunikation oder das Auslösen einer Exception im Thread.

`void l4_thread_control_start(` `  void)`	Startet eine Thread-Control-Sequenz.
`void l4_thread_control_pager(` `  l4_cap_idx_t pager)`	Setzt den Pager.
`void l4_thread_control_exc_handler(` `  l4_cap_idx_t exc_handler)`	Setzt den Exception-Handler.

`void l4_thread_control_bind(`
`  l4_utcb_t  *thread_utcb,`
`  l4_cap_idx_t task)`
	Bindet den Thread an den übergebenen Task unter der übergebenen UTCB-Adresse.

`l4_msgtag_t l4_thread_control_commit(` `  l4_cap_idx_t thread)`	Beendet die Thread-Control-Sequenz und vollzieht die Änderungen für den übergebenen Thread.

UTCB	
`l4_utcb_t * l4_utcb(void)`	Gibt die Adresse des UTCB des aktuellen Threads zurück.
`l4_msg_regs_t * l4_utcb_mr(void)`	Gibt einen Zeiger auf den Nachrichtenregisterblock im UTCB des aktuellen Threads zurück.
Mapping	
`l4_fpage_t l4_fpage(unsigned long address, unsigned int size, unsigned char rights)`	Erstellt eine Flexpage an der spezifizierten Startadresse mit der übergebenen Länge (bemessen in der Größe des Logarithmus zur Basis 2) und den übergebenen Rechten.
`l4_fpage_t l4_obj_fpage(l4_cap_idx_t obj, unsigned int order, unsigned char rights)`	Erstellt eine Flexpage für das spezifizierte Kernelobjekt mit der übergebenen Anzahl an Capabilities (bemessen in der Größe des Logarithmus zur Basis 2) und den übergebenen Rechten.
`l4_umword_t l4_map_control(l4_umword_t spot, unsigned char cache, unsigned grant)`	Erstellt das initiale Maschinenwort für ein einzubindendes Item im Memory Space.
`l4_umword_t l4_map_obj_control(l4_umword_t spot, unsigned grant)`	Erstellt das initiale Maschinenwort für ein einzubindendes Item im Object Space (Capability-Tabelle für Kernelobjekte).
`l4_msgtag_t l4_task_map(l4_cap_idx_t dst_task, l4_cap_idx_t src_task, l4_fpage_t const snd_fpage, l4_addr_t snd_base)`	Bindet die spezifizierte Flexpage des ursprünglichen Tasks in den Adressbereich des Zieltasks ein.

IPC	

`l4_msgtag_t l4_msgtag(`
`  long label,`
`  unsigned words,`
`  unsigned items,`
`  unsigned flags)`
	Erstellt ein Message-Tag mit einem benutzerdefinierten Label, der Anzahl der nicht typisierten Maschinenwörter, der Anzahl der typisierten Elemente (z.B. Flexpages) sowie weiteren Flags.

`l4_msgtag_t l4_ipc_call(`
`  l4_cap_idx_t object,`
`  l4_utcb_t *utcb,`
`  l4_msgtag_t tag,`
`  l4_timeout_t timeout)`
	Vollzieht einen IPC-Aufruf an ein Kernelobjekt mit den Inhalten des übergebenen UTCB, einem Message-Tag sowie einem definierten Timeout und wartet auf eine Rückmeldung des Empfängers.

`l4_msgtag_t l4_ipc_send(`
`  l4_cap_idx_t dest,`
`  l4_utcb_t *utcb,`
`  l4_msgtag_t tag,`
`  l4_timeout_t timeout)`
	Sendet eine Nachricht an ein Kernelobjekt mit den Inhalten des übergebenen UTCB, einem Message-Tag sowie einem definierten Timeout und wartet **nicht** auf eine Rückmeldung des Empfängers.

`l4_msgtag_t l4_ipc_receive(`
`  l4_cap_idx_t object,`
`  l4_utcb_t *utcb,`
`  l4_timeout_t timeout)`
	Wartet auf den Empfang einer Nachricht von einem definierten Absender.

`l4_msgtag_t l4_ipc_wait(`
`  l4_utcb_t *utcb,`
`  l4_umword_t *label,`
`  l4_timeout_t timeout)`
	Wartet auf den Empfang einer Nachricht von einem beliebigen Absender und speichert die Absenderinformationen im übergebenen Label.

`l4_msgtag_t l4_ipc_reply_and_wait(`
`  l4_utcb_t *utcb,`
`  l4_msgtag_t tag,`
`  l4_umword_t *label,`
`  l4_timeout_t timeout)`
	Sendet dem Absender der vorangegangenen Nachricht eine Rückmeldung und wartet anschließend (erneut) auf den Empfang einer Nachricht von einem beliebigen Absender (prädestiniert für den Einsatz in einer Serverumgebung).

Scheduler	

`l4_sched_param_t l4_sched_param(`
`  unsigned prio,`
`  l4_cpu_time_t quantum`
`  L4_DEFAULT_PARAM(0))`
	Instanziiert einen neuen Satz an Scheduling-Parametern, bestehend aus Priorität und Quantum.

`l4_msgtag_t l4_scheduler_run_thread(`
`  l4_cap_idx_t scheduler,`
`  l4_cap_idx_t thread,`
`  l4_sched_param_t const *sp)`
	Übergibt den Thread *thread* mit den Scheduling-Parametern *sp* an den Scheduler *scheduler*.

Literatur

[1] Luca Abeni und Giorgio C. Buttazzo. *Integrating multimedia applications in hard real-time systems*. Dez. 1998. ISBN: 978-0-8186-9212-3.

[2] Manuel Bernard, Dr. Christian Buckl, Volkmar Döricht u. a. *Mehr Software (im) Wagen: Informations- und Kommunikationstechnik (IKT) als Motor der Elektromobilität der Zukunft*. http://www.projekt-race.de/upload/downloads/ikt2030de-gesamt.pdf. März 2011. (Besucht am 10. 10. 2014).

[3] Giorgio C. Buttazzo. *Hard real-time computing systems. Predictable scheduling algorithms and applications*. 3rd. 2011. ISBN: 978-1-4614-0675-4.

[4] TU Dresden. *Fiasco.OC & L4 Runtime Environment (L4Re)*. http://os.inf.tu-dresden.de/L4Re/doc/index.html. (Besucht am 30. 08. 2014).

[5] TU Dresden. *L4 Runtime Environment: examples/clntsrv/clntsrv.cfg*. http://os.inf.tu-dresden.de/L4Re/doc/examples_2clntsrv_2clntsrv_8cfg-example.html. (Besucht am 30. 08. 2014).

[6] TU Dresden. *L4 Runtime Environment: examples/sys/utcb-ipc/main.c*. http://os.inf.tu-dresden.de/L4Re/doc/examples_2sys_2utcb-ipc_2main_8c-example.html. (Besucht am 30. 08. 2014).

[7] TU Dresden. *The L4 microkernel family - Overview*. http://os.inf.tu-dresden.de/L4/overview.html. Juli 2014. (Besucht am 30. 08. 2014).

[8] Björn Döbel. *Microkernel-based Operating Systems - Introduction*. http://os.inf.tu-dresden.de/Studium/KMB/WS2013/01-Introduction.pdf. Okt. 2013. (Besucht am 30. 08. 2014).

[9] Georgia Giannopoulou u. a. *Scheduling of mixed-criticality applications on resource-sharing multicore systems*. http://www.tik.ethz.ch/~phuang/resource/EMSOFT13.pdf. Aug. 2013. (Besucht am 10. 10. 2014).

[10] Steve Heath. *Embedded Systems Design*. 2nd. 2002, 2ff. ISBN: 978-0-7506-5546-0.

[11] Hermann Härtig u. a. *The performance of µ-kernel-based systems*. Okt. 1997, S. 66–77. ISBN: 0-89791-916-5.

[12] Frank Mehnert, Jan Glauber und Jochen Liedtke. *Fiasco Kernel Debugger Manual*. http://os.inf.tu-dresden.de/fiasco/doc/jdb.pdf. Feb. 2008. (Besucht am 04. 09. 2014).

[13] Andrew S. Tanenbaum. *Modern operating systems*. 3rd. 1992, S. 62–65. ISBN: 0-13-813459-6.

[14] Carnegie Mellon University. *CMU CS Project Mach Home Page*. http://www.cs.cmu.edu/afs/cs/project/mach/public/www/mach.html. Feb. 1997. (Besucht am 30. 08. 2014).