Integration einer Mehrprozessorunterstützung in das Java-Betriebssystem JX

Diplomarbeit im Fach Informatik

vorgelegt von

Meik Felser

geboren am 15. Juni 1976 in Nürnberg

Angefertigt am

Lehrstuhl für Informatik 4
Verteilte Systeme und Betriebssysteme
Friedrich-Alexander-Universität Erlangen-Nürnberg

Betreuer:	**Prof. Dr. rer. nat. Fridolin Hofmann** **Dipl. Inf. Michael Golm**

Beginn der Arbeit:	02. Januar 2001
Abgabe der Arbeit:	02. Juli 2001

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Erlangen, den 02. Juli 2001, ________________________________

Meik Felser

Kurzfassung

Im Rahmen dieser Arbeit wurde das Java-Betriebssystem JX um eine Unterstützung für Intel-Multiprozessorsysteme erweitert. Dabei wurden drei Bereiche verändert:

1. Kritische Datenstrukturen wurden durch gegenseitigen Ausschluss vor gleichzeitiger Veränderung geschützt.
2. Die Interruptverwaltung wurde an die Gegebenheiten eines Multiprozessorsystems angepasst.
3. Die Verwaltung der Aktivitätsträger (Threads) wurde auf die Verwendung mehrerer Prozessoren ausgelegt.

Dem letzten Punkt wurde besondere Aufmerksamkeit geschenkt. So wurde ein flexibles Schedulingkonzept entwickelt und implementiert, welches nicht nur die Verwaltung mehrerer Prozessoren zulässt, sondern auch die Anforderung von JX erfüllt: den Anwendungen ein unabhängiges Scheduling zu ermöglichen. Der Schwerpunkt lag dabei auf der Definition von Schnittstellen, die nötig sind, um eine flexible Realisierung eines Schedulersystems zu ermöglichen.

Das Konzept sieht zwei Schedulingebenen vor. Eine Ebene teilt die Rechenzeit den Anwendungen zu, die zweite Ebene verteilt die Rechenzeit innerhalb der Anwendung. Dabei werden Schedulerobjekte eingesetzt, die vollkommen in Java implementiert sind, um eine leichte Veränderung der Schedulingstrategie zuzulassen. Um die Möglichkeiten des Konzepts zu zeigen, wurde exemplarisch ein Schedulingsystem auf Basis der Round-Robin-Strategie implementiert.

Um Interprozesskommunikation zu realisieren, werden in JX so genannte *Portale* eingesetzt, mit deren Hilfe Methoden anderer Anwendungen aufgerufen werden können. Da die Anwendungen voneinander isoliert sind, ist ein Portalaufruf mit einer Threadumschaltung zum Anbieter verbunden. Im Laufe dieser Arbeit wurde das Schedulingsystem in den Portalmechanismus integriert, sodass sowohl der Anbieter eines Portals als auch der Nutzer mehr Kontrolle über den Ablauf erhalten können.

Abstract

The purpose and intent of this diploma-thesis was to extend the Java operating system JX to enable support of Intel multiprocessor systems. To accomplish this, three areas were considered:

1. Critical data structures were protected against simultaneous modification by means of mutal exclusion.
2. Interrupt handling was adapted to satisfy the needs of a multiprocessor system.
3. Administration of threads was re-designed to use the advantages of multiple processors.

Particular attention was given to the last area. A flexible scheduling concept was developed and implemented to allow the administration of multiple processors. The main focus was to define necessary interfaces to enable a flexible implementation of the scheduling system.

This concept splits CPU scheduling into two levels. The first assigns the CPU to the applications while the second level distributes the computing time within an application. The scheduler objects are implemented entirely in Java as this allows for easy modification of the scheduling strategy. In order to demonstrate the functionality of this concept a simple round robin scheduling system was implemented.

JX uses so-called *portals* to effect inter-process communication. A portal can be used to invoke methods of other applications. The isolation of the different applications requires a context switch to the portal provider. Therefore the Scheduler was integrated in the portal mechanism to allow both, the provider of a portal and the user, to retain full control of the CPU time and thread activation.

Inhaltsverzeichnis

1 Einleitung **1**

2 Das Betriebssystem JX **3**

2.1 Überblick . . . 3

2.2 Domains und Portale . . . 3

2.3 Interrupts . . . 5

3 Multiprozessorsysteme **6**

3.1 Einführung . . . 6

3.2 Synchronisation in Multiprozessorsystemen . . . 7

3.3 Intel-Multiprozessorsysteme . . . 9

3.3.1 Überblick . . . 9

3.3.2 Prozessor-Kopplung . . . 10

3.3.3 Monoprozessor-Kompatibilität . . . 13

3.3.4 Konfigurationstabelle . . . 14

3.3.5 Identifikation eines Prozessors . . . 17

3.3.6 Aktivierung eines Prozessors . . . 18

3.4 Implementierung . . . 21

3.5 Zusammenfassung . . . 22

4 Interruptsystem **23**

4.1 Konfiguration der I/O-APICs . . . 23

4.2 Lokale APICs . . . 25

4.2.1 Konfiguration . . . 25

4.2.2 Interprozessorinterrupts . . . 25

4.2.3 Lokaler Timer . . . 26
4.3 Implementierung . . . 26
4.4 Zusammenfassung . . . 28

5 Scheduler **29**
5.1 Überblick . . . 29
5.2 Ausgangssituation . . . 30
5.3 Anforderungen und Zielsetzung . . . 31
5.4 Konzept . . . 31
5.4.1 Zweistufiges Grundkonzept . . . 31
5.4.2 Unterstützung für Multiprozessorsysteme . . . 33
5.4.3 Gemeinsames Schedulingobjekt für mehrere CPUs . . . 34
5.4.4 Externes Scheduling . . . 35
5.5 Kommunikation mit Schedulern . . . 36
5.5.1 Einführung . . . 37
5.5.2 Benachrichtigungsmöglichkeiten . . . 37
5.5.3 Interaktion mit Highlevel-Schedulern . . . 39
5.5.4 Funktionsaufrufe über Domaingrenzen . . . 41
5.5.5 Geschwindigkeitsvergleich verschiedener Konzepte . . . 44
5.5.6 Konfiguration der Benachrichtigungen . . . 45
5.5.7 Implementierung . . . 47
5.5.8 Alternative Interaktion mit Hilfe von Ereignislisten . . . 49
5.6 Timer . . . 50
5.6.1 Zielsetzung . . . 51
5.6.2 Timer in Multiprozessorsystemen . . . 51
5.6.3 Globale Zeitscheibe . . . 52
5.6.4 Individuelle Zeitscheibe pro Domain . . . 52
5.6.5 Benachrichtigungen . . . 53
5.6.6 Bewertung . . . 54
5.6.7 Ausblick . . . 54
5.7 Bewertung . . . 54
5.8 Zusammenfassung . . . 55

6 Portale **56**

6.1 Einführung . . . 56
6.2 Ausgangssituation . . . 57
6.3 Motivation . . . 57
6.3.1 Kooperatives Scheduling . . . 58
6.3.2 Blockierende Portalmethoden . . . 58
6.4 Ziele . . . 59
6.5 Integration des Schedulers . . . 59
6.5.1 Kontrolle des Portalnutzers . . . 59
6.5.2 Kontrolle des Dienstanbieters . . . 61
6.6 Multiprozessorunterstützung . . . 61
6.6.1 Einhaltung der Ressourcenbeschränkung . . . 61
6.6.2 Mehrere Threads pro Portal . . . 62
6.7 Optimierung . . . 63
6.8 Geschwindigkeitsvergleich . . . 64
6.9 Ausblick . . . 65
6.10 Zusammenfassung . . . 66

7 Zusammenfassung und Ausblick **68**

7.1 Zusammenfassung . . . 68
7.2 Bewertung und Ausblick . . . 69

A Kern-Schnittstellen **71**

A.1 Lowlevel-Scheduler . . . 71
A.1.1 Das Interface `LowlevelScheduler` . . . 71
A.1.2 Das Portal `LLSchedulerSupport` . . . 73
A.2 Highlevel-Scheduler . . . 75
A.2.1 Das Basisinterface `HighLevelScheduler` . . . 75
A.2.2 Optionale Interfaces für Highlevel-Scheduler . . . 77
A.2.3 Das Portal `HLSchedulerSupport` . . . 79
A.3 Das Portal `SMPCPUManager` . . . 81

Abbildungsverzeichnis

2.1 Das Konzept von JX . . . 4

3.1 Prozessor-Kopplung in Intel-MP-Systemen . . . 10
3.2 Interruptsystem im Monoprozessor-PC . . . 11
3.3 Interruptsystem in einem Intel-Multiprozessorsystem . . . 12
3.4 Interrupt Modi . . . 14
3.5 Zeigerstruktur und Konfigurationstabelle . . . 15
3.6 Pseudocode zum Aktivieren eines Prozessors . . . 19

5.1 Ausgangssituation: globaler Scheduler . . . 30
5.2 Zweistufiges Grundkonzept . . . 32
5.3 Zweistufiges Konzept für Multiprozessorsysteme . . . 33
5.4 Threadverwaltung in externer Domain . . . 36
5.5 Kommunikation zwischen den Schedulern am Beispiel eines Domainwechsels . . . 37
5.6 Funktionsweise eines Schedulerportals . . . 42
5.7 Realisierte Interfaces eines einfachen Round-Robin-Schedulers . . . 46
5.8 Kommunikation zwischen den Schedulern im Detail . . . 49
5.9 Beteiligte Threads bei einem Domainwechsel . . . 50
5.10 Ablauf bei alternativer Kommunikation am Beispiel eines Domainwechsels . . . 50
5.11 Konflikt zwischen globaler und domainspezifischer Zeitscheibe . . . 52
5.12 Timer mit Zustandssicherung . . . 53

6.1 Portalaufruf . . . 56
6.2 Portalmechanismus (vor dieser Arbeit) . . . 57
6.3 Portalmechanismus mit Benachrichtigung des Schedulers einer aufrufenden Domain . . . 60
6.4 Portalaufruf mit inkompatiblen Prozessornutzungsrechten . . . 61

Tabellenverzeichnis

3.1 Reihenfolge bei der Suche nach der Zeigerstruktur 15
3.2 Die wichtigsten Informationen in der Zeigerstruktur 15
3.3 Einträge in der Konfigurationstabelle und ihre wichtigsten Informationen 16

4.1 Die Umsetzungstabelle eines I/O-APICs 24

5.1 Mögliche Benachrichtigungen eines Highlevel-Schedulers 38
5.2 Geschwindigkeitsvergleich zwischen Portal und Schedulerportal 43
5.3 Geschwindigkeitsvergleich zwischen verschiedenen Schedulingkonzepten 44
5.4 Geschwindigkeitsgewinn durch "Abschalten" einer Benachrichtigung 46
5.5 Geschwindigkeitsvergleich zwischen Methodenaufruf und *inlined* Portal 49
5.6 Zeitbedarf für das Einstellen eines neuen Timer-Intervalls 52

6.1 Geschwindigkeit des optimierten Portalmechanismus 63
6.2 Zeit für einen Portalaufruf bei verschiedenen Konfigurationen 64
6.3 Geschwindigkeitsvergleich zwischen einer Portalausführung auf der aktuellen CPU und auf einer anderen CPU 65

1 Einleitung

Computersysteme mit mehr als einem Prozessor wurden bislang meist im Bereich des Hochleistungsrechnens eingesetzt. Jedoch finden Multiprozessorarchitekturen zunehmend auch im allgemeinen Bereich Anwendung. Vor allem kleine Multiprozessorsysteme bis zu 8 Prozessoren werden gerne als Server oder sogar als Arbeitsplatzrechner eingesetzt.

Häufig findet man dabei Intel-Multiprozessorsysteme vor. Diese Systeme bieten den Vorteil, dass sie völlig kompatibel zu einem PC/AT-Monoprozessorsystem sind, solange nur ein Prozessor verwendet wird. Außerdem begünstigen vor allem die sinkenden Hardware-Preise den vielseitigen Einsatz von Multiprozessorsystemen. So bekommt man heute ein günstiges Dual-Prozessorsystem bereits um einen Betrag, den man noch vor wenigen Jahren für ein System mit nur einem Prozessor investieren musste.

Um die Möglichkeiten eines Multiprozessorrechners zu nutzen, benötigt man die Unterstützung des Betriebssystems. Die zunehmende Verbreitung von Multiprozessorsystemen führt dazu, dass immer mehr Betriebssysteme Unterstützung für diese Systeme anbieten.

Das Ziel dieser Arbeit war, das Java-Betriebssystem JX so zu erweitern, dass Intel-Multiprozessorsysteme unterstützt werden. Das JX-System ist ein Forschungsprojekt, an dem ein Speicherschutzkonzept untersucht wird, welches auf den Sicherheitskonzepten von Java basiert – und somit in Software realisiert ist. Mit dem Projekt soll gezeigt werden, dass diese Art der Absicherung einem hardwarebasierten Speicherschutz nicht unterlegen ist.

Um Multiprozessorsysteme zu unterstützen, muss unter anderem die Interruptverarbeitung entsprechend angepasst werden. Dabei muss auf die besondere Hardware eines solchen Systems eingegangen werden. Ein Ziel dabei ist, einen Interrupt einem beliebigen Prozessor zuordnen zu können. Nutzen kann man diese Fähigkeit beispielsweise bei einem HTTP-Server, der über zwei Netzwerkkarten und zwei Prozessoren verfügt. Dort könnte jeder Prozessor je eine Netzkarte verwalten.

Neben der Interruptverwaltung muss auch die Thread-Verwaltung an ein Multiprozessorsystem angepasst werden. Da das vorhandene Schedulingsystem nur einen Prozessor unterstützte und zudem auch als Provisorium gedacht war, ist ein weiteres Ziel dieser Arbeit die Entwicklung eines flexiblen Schedulingkonzepts.

Ein Punkt, der dabei besonders beachtet wurde ist, dass jede Anwendung (Domain) ihre Aktivitätsträger selbst individuell verwalten kann. Es wird deshalb gefordert, dass die Scheduler in Java implementiert werden, um die Schedulingstrategie leicht verändern zu können. Außerdem soll die Realisierung eines Schedulers durch verschiedene Konfigurationsmöglichkeiten unterstützt werden.

Allgemein sollte untersucht werden, welche Schnittstellen nötig sind, um eine flexible Implementierung eines Schedulers in Java zu ermöglichen. Die Tauglichkeit für Multiprozessorsysteme war dabei besonders zu berücksichtigen.

Struktur der Arbeit

Nach dieser Einleitung wird in Kapitel 2 zunächst ein kurzer Überblick über das Java-Betriebssystem JX gegeben. Dieses System bildete die Basis dieser Arbeit.

Kapitel 3 beschäftigt sich anschließend mit den Grundlagen eines Intel-Multiprozessorsystems. Zuerst wird der Aufbau eines solchen Systems beschrieben, wobei auch auf Unterschiede zu Monoprozessorsystemen eingegangen wird. Danach werden einige Aspekte zur Ansteuerung einzelner Prozessoren gegeben. Abschließend wird kurz auf die Implementierung eingegangen, die zur Erkennung und Aktivierung eines Multiprozessorsystems erstellt wurde.

In Kapitel 4 wird genauer auf das Interruptsystem eines Intel-MP[1]-Systems eingegangen. Dabei wird auch erläutert, welche Anpassungen an JX vorgenommen werden mussten und welche Möglichkeiten in einer Multiprozessorumgebung zur Verfügung stehen.

Das fünfte Kapitel ist dem Schedulingkonzept gewidmet, welches im Rahmen dieser Arbeit entworfen wurde. Nachdem das Konzept vorgestellt wurde, wird auf die Interaktion der Scheduler eingegangen. Anschließend wird ein Konzept vorgestellt, welches die Realisierung individueller Zeitscheiben für jede Domain zulässt.

Im darauffolgenden Kapitel wird auf die Möglichkeiten der Interprozesskommunikation eingegangen, die JX zur Verfügung stellt. Dabei wird zum einen der Kommunikationsmechanismus so erweitert, dass ein Scheduler die vollständige Kontrolle, sowohl über die Rechenzeit als auch über die Threads einer Anwendung (Domain), behalten kann. Zum anderen wird auf Besonderheiten eingegangen, die es in Bezug auf Multiprozessorsysteme zu berücksichtigen gilt.

Das letzte Kapitel gibt eine Zusammenfassung der erreichten Ergebnisse sowie einen Ausblick auf Erweiterungen.

In Anhang A werden anschließend die Schnittstellen beschrieben, die ein Scheduler zur Verfügung stellen muss bzw. kann. Außerdem werden Methoden vorgestellt, die der Kern den Schedulern zur Verfügung stellt, um miteinander zu kommunizieren.

[1] Im Folgenden wird der Ausdruck *Multiprozessor* auch durch MP abgekürzt.

2 Das Betriebssystem JX

Im folgenden Kapitel soll ein Überblick über das Java-Betriebssystem JX [GK00, GKB01] gegeben werden.

2.1 Überblick

JX ist ein experimentelles Betriebssystem, welches vollständig in Java geschrieben ist. Die einzige Ausnahme hiervon ist der Mikrokern, auf dem das Betriebssystem aufbaut: dieser ist in C und teilweise in Assembler implementiert.

Die Besonderheit von JX liegt in der Absicherung der einzelnen Anwendungen voneinander. Hierbei setzt JX nicht auf einen adressraum-basierten Schutz – in dem jede Anwendung ihren eigenen Adressraum hat, der durch Hardware-Unterstützung (MMU) gesichert ist –, sondern auf die Sicherheitskonzepte, die durch die Programmiersprache Java zugesichert werden.

Da Java keine Zeigerarithmetik (wie C oder C++) zulässt, sondern nur mit typisierten Referenzen arbeitet, ist es durch ein korrektes Java-Programm nicht möglich beliebige Speicherstellen zu adressieren. Die Korrektheit eines Java-Programms bezüglich des Speicherzugriffs, d. h. die Gültigkeit der Referenzen, wird durch den Java-Compiler bzw. einen Java-Bytecode-Verifier [Alt01] sichergestellt.

Durch diese Absicherung des Speichers vor unbefugten Zugriffen ist es möglich, für alle Anwendungen einen gemeinsamen Adressraum zu verwenden. Die Trennung der Anwendungen erfolgt durch die Kapselung in so genannte *Domains*, welche bis auf die *Domain Zero* (siehe nächster Abschnitt) ausschließlich in Java implementiert sind.

2.2 Domains und Portale

Bei *Domains* handelt es sich um Ausführungsumgebungen, die voneinander unabhängig und entkoppelt sind. Sie enthalten den Code der auszuführenden Komponenten und stellen jeweils einen isolierten Typraum dar. Das bedeutet, dass Datentypen, die in einer Domain gelten, nicht zwangsläufig auch in jeder anderen Domain Gültigkeit haben.

Im Großen und Ganzen könnte man Domains mit Prozessen in UNIX vergleichen. Einer Domain sind bestimmte Ressourcen zugeordnet, sie hat einen eigenen Heap und eigene Aktivi-

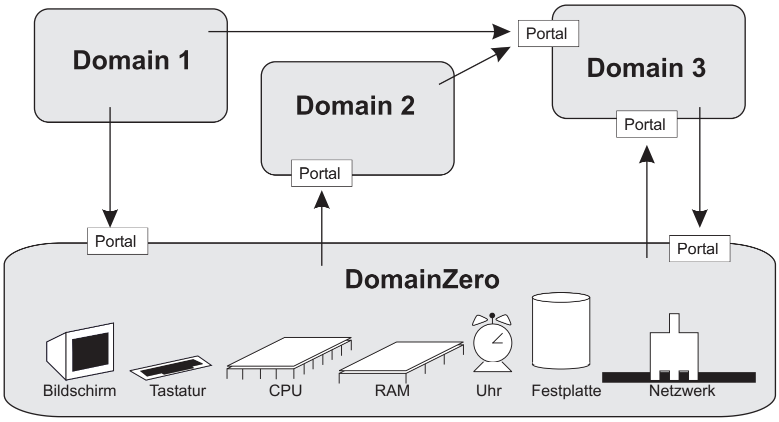


Abbildung 2.1: Das Konzept von JX
(Nach: [GKB01])

tätsträger (Threads). Die Speicherverwaltung einer Domain – inklusive Garbage Collector – kann unabhängig von den anderen Domains implementiert sein.

Portale

Um untereinander zu kommunizieren, können Domains so genannte *Portale* zur Verfügung stellen (siehe Kapitel 6). Diese Portale stellen sich als Java-Objektreferenz dar und stellen die Methoden zur Verfügung, welche von der erzeugenden Domain freigegeben wurden. Um eine Methode einer anderen Domain aufzurufen, wird daher eine Portal-Referenz benötigt, welche man entweder von einen Namensservice oder direkt von einer anderen Domain erhalten kann.

Ruft eine Domain eine Methode an einem Portal auf, so erfolgt eine Umschaltung auf einen Thread der Zieldomain. Dieser Thread wird mit dem Portal bei dessen Erzeugung verknüpft und führt die Methoden aus, die über das Portal angeboten werden.

Domain Zero

Der Kern des Betriebssystem wird durch die *Domain Zero* repräsentiert. Dabei handelt es sich um eine besondere Domain, die als einziges nicht in Java, sondern in C und Assembler, implementiert ist. Von ihr werden Portale exportiert, die den Zugriff auf die Hardware des Systems erlauben.

Speziell für die Domain Zero existiert eine besondere Portal-Art: so genannte *inlined* Portale. Im Gegensatz zu normalen Portalen findet hier bei einem Aufruf keine Thread-Umschaltung statt, sondern die Methode wird direkt vom aufrufenden Thread ausgeführt. Der Vorteil dieser besonderen Portal-Art ist daher ein geringerer Overhead. Allerdings sind solche Portale nur für Methoden geeignet, die keine Objekte der bereitstellenden Klasse verwenden,[1] da jede Domain isoliert ist und somit nicht auf Objekte anderer Domains zugegriffen werden kann.

[1] Bisher sind sie nur für Methoden der Domain Zero einsetzbar.

2.3 Interrupts

Wird in einer Domain zum Beispiel ein Gerätetreiber implementiert, so muss meist auf einen Interrupt reagiert werden. Hierzu muss von der Domain eine spezielle Methode – der so genannte Interrupt-Handler – bereitgestellt werden, die beim Auftreten des entsprechenden Interrupts ausgeführt wird.

Der Interrupt-Handler muss zunächst beim Kern – der Domain Zero – registriert werden. Bei der Registrierung wird zur Ausführung des Codes ein eigener Thread erzeugt. Ein Interrupt-Handler ist daher mit einem normalen Portal vergleichbar, unterliegt jedoch zusätzlichen Anforderungen, zum Beispiel an die Ausführungszeit und die verwendeten Methoden.

3 Multiprozessorsysteme

Das folgende Kapitel soll einen Überblick über Multiprozessorsysteme geben. Dabei werden zunächst grundlegende Konzepte und verschiedene Möglichkeiten der Charakterisierung vorgestellt. Anschließend wird der Aufbau von Intel-Multiprozessorsystemen vorgestellt.

3.1 Einführung

Unter den Begriff Multiprozessorsysteme fasst man alle Rechnersysteme zusammen, die über mehr als einen Prozessor verfügen. Es gibt jedoch unterschiedliche Arten solcher Systeme, die im Folgenden in Gruppen eingeteilt werden. Dabei werden verschiedene Konzepte angesprochen, die als Charakteristiken dienen können.

Eine grundlegende Unterscheidung kann nach der Art der Prozessoren getroffen werden [Hof89]:

- homogene Systeme
  Alle Prozessoren sind vom gleichen (oder ähnlichen) Typ.
- heterogene Systeme
  Bei den Prozessoren des Systems handelt es sich um unterschiedliche Typen. Meist handelt es sich um Spezialprozessoren, die für eine bestimmte Aufgabe optimiert sind, wie beispielsweise Vektorprozessoren oder spezielle E/A-Prozessoren.

Eine andere Einteilung erfolgt nach der Art des Speicherkopplung. Dabei werden drei Klassen unterschieden [Ras87]:

- **U**niform **M**emory **A**ccess (UMA)
  Der gesamte verfügbare Arbeitsspeicher ist allen Prozessoren zugänglich (gemeinsamer Adressraum). Die Kosten für einen Zugriff sind unabhängig von Prozessor und Speicherstelle.
- **N**o **U**niform **M**emory **A**ccess (NUMA)
  Jedem Prozessor ist ein eigener Speicher zugeteilt. Die Nutzung durch einen anderen Prozessor ist zwar im Allgemeinen möglich, allerdings sind die Kosten für eine Speicheroperation abhängig von der Speicherposition (Zugriffe auf eigenen Speicher sind billiger als Zugriffe auf fremde Speicherbereiche).

- **NO R**emote **M**emory **A**ccess (NORMA)
  In dieser Klasse verfügen die Prozessoren über keinen gemeinsamen Adressraum, sodass kein direkter Zugriff auf den Speicher eines anderen Prozessors möglich ist.

Die Kopplung der einzelnen Prozessoren beeinflusst die Art der Kommunikation:

- Kommunikation mit Hilfe eines gemeinsamen Speichers (shared memory)
  Hierbei werden die Informationen an einer definierten Stelle im gemeinsamen Speicher abgelegt. Dort können sie von anderen Prozessoren gelesen werden.
- Kommunikation durch Nachrichtenversand (message passing)
  Hierbei werden explizit Nachrichten zwischen den Prozessoren verschickt, um Informationen auszutauschen. Bei Systemen ohne gemeinsamen Adressraum (NORMA) ist dies die einzige Möglichkeit der Kommunikation.

Speziell bei Systemen mit gemeinsamen Speicher, die diesen auch zur Koordination und Kommunikation verwenden, kann der Speicherbus oft zum Engpass werden. Deshalb wird jeder Prozessor mit einem Pufferspeicher (Cache) ausgestattet, der einen kleinen Teil des Hauptspeichers aufnehmen kann und einen besonders schnellen Zugriff erlaubt. Diese zusätzliche Indirektionsstufe führt jedoch zu einem Problem: verändert ein Prozessor eine Speicherstelle, die ein anderer Prozessor gepuffert hat, so sind die Daten im Pufferspeicher ungültig.

Die Art dieses Problem zu lösen, lässt eine Einteilung in zwei Gruppen zu:

- cachekohärente Systeme
  Diese sorgen automatisch dafür, dass die Daten in den Pufferspeichern der einzelnen Prozessoren gleich (kohärent) gehalten werden – der Cache ist somit völlig transparent.
- Systeme ohne Cachekohärenz
  Hierbei muss der Benutzer selbst darauf achten, dass Diskrepanzen zwischen dem Inhalt des Caches und dem Hauptspeicher auftreten können. Zur Vermeidung kann er beispielsweise einen Bereich des Speichers – der zum Informationsaustausch verwendet wird – grundsätzlich nicht puffern lassen.

3.2 Synchronisation in Multiprozessorsystemen

Werden Daten von mehreren Threads gemeinsam verwendet, so muss der Zugriff darauf koordiniert werden. In einer Multiprozessorumgebung werden darüberhinaus auch Mechanismen benötigt, um parallel ablaufende Aktivitätsträger zu koordinieren. Eine Klassifikation von Koordinierungsproblemen gibt [Hof90], dort werden auch einige Basismechanismen vorgestellt. Hier soll diese Thematik nur kurz angesprochen werden.

Häufig werden Sperren eingesetzt um einen kritischen Abschnitt vor gleichzeitigem Zugriff zu schützen. Die Anforderungen an Koordinationsmechanismen sind je nach Zielplattform unterschiedlich. Während in einer Monoprozessorumgebung ein Thread, der auf einen gesperrten Bereich zugreifen möchte, die Kontrolle über den Prozessor aufgeben muss, kann es in MP-Systemen sinnvoll sein, aktiv zu warten (busy wait), bis der kritische Abschnitt frei ist.

Während des "Wartens" wird meist der Zustand der Sperre überprüft. Dadurch kann jedoch eine erhebliche Busbelastung entstehen. In [MC91] werden mögliche Sperr-Algorithmen für Multiprozessorsysteme vorgestellt und bezüglich der Systembelastung verglichen. Daneben wird auch auf *Barrieren* für Multiprozessorsysteme eingegangen, mit deren Hilfe man mehrere parallel ablaufende Threads synchronisieren kann.

Die Verwendung von Sperren hat jedoch folgenden Nachteil: fällt der Prozessor, der die Sperre hält aus, so kann kein anderer mehr den kritischen Abschnitt betreten. Weiterhin kann es zu Verklemmungen kommen, wenn mehrere Sperren benötigt werden und diese in einer falschen Reihenfolge angefordert werden.

Als Alternative zum Einsatz von Sperren ist deshalb die nicht-blockierende Synchronisation zu nennen. Diese verzichtet auf den gegenseitigen Ausschluss bei kritischen Abschnitten, stattdessen werden niemals inkonsistente Zustände der gemeinsamen Daten erzeugt. Dies wird in der Regel dadurch realisiert, dass Berechnungen erst vollständig durchgeführt werden, bevor das Ergebnis in den Speicher geschrieben wird. Werden die Ausgangsdaten während der Bearbeitung verändert, so wird das Ergebnis verworfen und die Berechnung erneut durchgeführt. Voraussetzung dafür sind jedoch atomare Operationen wie z. B. die *Compare-and-Swap* Operation[1].

Durch nicht-blockierende Synchronisation kann gewährleistet werden, dass durch den Ausfall eines Prozessors im kritischen Abschnitt nicht das gesamte System blockiert, da die anderen ihre Berechnungen oder Operationen auf den gemeinsamen Daten trotzdem durchführen können.[2] Nicht-blockierende Koordinationsmechanismen sind allerdings meist sehr aufwändig.

[1]Eine Compare-and-Swap Anweisung benötigt drei Parameter: eine Speicherstelle (durch einen Zeiger spezifiziert), einen alten und einen neuen Wert. Die Operation vergleicht atomar den Inhalt der Speicherstelle mit dem alten Wert; stimmen sie überein, so wird der neue Wert an die Speicherstelle geschrieben. Folgender Pseudocode veranschaulicht die Operation:

```
boolean CAS (int *data, int old, int new)
   BEGIN_ATOMIC
      if *data = old then
         *data ← new;
         return TRUE;
      else
         return FALSE;
      end if
   END_ATOMIC
end
```

[2]Tatsächlich garantieren nicht-blockierende Algorithmen lediglich, dass *einige* Prozesse ihre Operationen durchführen können. Wird garantiert, dass *alle* fehlerfreien Prozesse ihre Operation in endlicher Zeit durchführen können, so spricht man von "warte-freien" Algorithmen [Val95]. Der zusätzliche Aufwand den solche Algorithmen verursachen, steht jedoch in keinem Verhältnis zu den erreichbaren Zusicherungen.

In Betriebssystemen werden häufig verkettete Listen verwendet um Daten zu verwalten. In [Val95] wird ein Algorithmus vorgestellt, der einfach-verkettete Listen mit Hilfe der Compare-and-Swap Operation realisiert. Dabei wird das Einfügen an beliebiger Stelle und das Löschen beliebiger Elemente unterstützt. Der Algorithmus ist allerdings relativ komplex und aufwändig.

Meist werden jedoch keine so allgemeinen Listen benötigt. Um eine Warteschlange zu realisieren reicht oft eine Liste mit FIFO-Funktionalität. Eine nicht-blockierende Realisierung dieser Funktionen wird z. B. in [MS96] oder [TZ00] beschrieben. Die Komplexität sowie der Aufwand dieser Algorithmen ist deutlich geringer.

Im Rahmen dieser Arbeit wurde lediglich eine einfache Sperre, aufbauend auf einen "Spinlock", implementiert. Einfache Barrieren können mit Hilfe dieses Mechanismus in Java realisiert werden; eine Unterstützung durch den Kern ist hierfür nicht erforderlich.

3.3 Intel-Multiprozessorsysteme

Die folgenden Abschnitte sollen einen Überblick über den Aufbau eines MP-Systems nach Intel-Multiprozessor-Spezifikation [Int97b] geben. Dabei werden insbesonders die Gesichtspunkte beschrieben, die für ein Betriebssystem relevant sind. Es wird auf Unterschiede zum Monoprozessorsystem eingegangen, sowie die möglichen Konfigurationen und das Aktivieren der Prozessoren beschrieben. Anschließend wird kurz auf die Implementierung im Rahmen dieser Arbeit eingegangen.

3.3.1 Überblick

Ein Intel-Multiprozessorsystem zeichnet sich primär dadurch aus, dass Prozessoren zum Einsatz kommen, die den Intel-Architektur Befehlssatz [Int99a] unterstützen. Vom Hersteller Intel kommen deshalb Prozessoren ab dem i486 in Frage.

Da es sich um ein Multiprozessorsystem handelt, sind mindestens zwei solcher Prozessoren im System vorhanden. Im Normalfall handelt es sich dabei um Prozessoren des gleichen Modells und gleicher Taktfrequenz. Die Intel-MP-Spezifikation [Int97b] sieht aber auch Fälle vor, bei denen verschiedene Modelle – sogar mit unterschiedlichen Taktfrequenzen – eingesetzt werden. So könnte zum Beispiel ein 486/66 mit einem Pentium 100 Prozessor zusammen ein Intel-MP-System bilden.

In einem Intel-MP-System können maximal 60 Prozessoren eingesetzt werden. Systeme mit mehr als 15 Prozessoren sind jedoch selten, da zusätzlicher Hardwareaufwand nötig ist, wie in Abschnitt 3.3.5 kurz angesprochen wird.

Eine Besonderheit der Intel-MP-Systeme ist die PC/AT-Kompatibilität. Das heißt, dass das System völlig kompatibel zu einem Standard IBM-kompatiblen (Monoprozessor-) PC ist. Dies ist aber nur dann der Fall, wenn nur ein Prozessor in Betrieb ist. Abschnitt 3.3.3 zeigt die Realisierung dieses Verhaltens.

Intel-Multiprozessorsysteme gehören zur Kategorie der so genannten SMP-Systeme (symmetric multiprocessing). Im Gegensatz zu asymmetrischen Systemen, bei denen einige Prozessoren dadurch ausgezeichnet sind, dass nur sie Zugriff auf spezielle Ressourcen haben, sind in symmetrischen Systemen alle Prozessoren gleichberechtigt.

Dies gilt insbesondere für den Zugriff auf den Speicher. Es liegt ein gemeinsamer Adressraum vor, in dem jeder Prozessor eine Speicherstelle unter derselben Adresse ansprechen kann. Es handelt sich somit um eine UMA-Architektur, da jeder Prozessor auf den gesamten Hauptspeicher zugreifen kann und der Speicher auch nicht einem bestimmten Prozessor zugeordnet ist.

Neben dem Speicher ist auch der Zugriff auf die E/A-Geräte allen Prozessoren möglich. Darunter fällt – neben den Ports – auch die Verwaltung der Interrupts, mit deren Hilfe E/A-Geräte mit dem Prozessor kommunizieren. Im folgenden Abschnitt wird darauf genauer eingegangen.

3.3.2 Prozessor-Kopplung

Die Kopplung der Prozessoren erfolgt – wie in Abbildung 3.1 dargestellt – zum einen über den Speicherbus, zum anderen sind die Prozessoren über einen Interrupt-Bus – den so genannten *Interrupt Controller Communications* (ICC) Bus – miteinander verbunden.

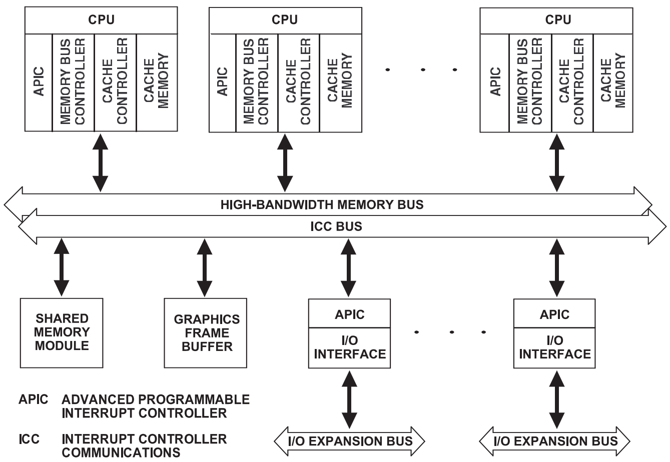


Abbildung 3.1: Prozessor-Kopplung in Intel-MP-Systemen
(Quelle: [Int97b])

Speicherbus

Die Kopplung über den Speicherbus ist für das Betriebssystem transparent. Die Hardware des Systems hat dafür Sorge zu tragen, das alle Caches kohärent sind. Dazu wird von Intel-Prozessoren das MESI-Protokoll implementiert. Dabei wird jede Cache-Zeile mit einer Markierung versehen, die angibt, ob der Inhalt verändert wurde (**M**odified), exklusiv (**E**xclusive) oder gemeinsam (**S**hared) genutzt wird oder ob die Zeile ungültig (**I**nvalid) ist. Durch dieses Protokoll werden Speicheränderungen – sowohl durch andere Prozessoren als auch durch

E-/A-Geräte – bekannt gemacht, um daraufhin die Markierung der Cache-Zeile entsprechend zu ändern. Genauere Informationen über das MESI-Cachekohärenzprotokoll findet man zum Beispiel in [Han93].

Um eine atomare Änderung im Speicher vorzunehmen, stellen Intel-Prozessoren das LOCK-Präfix zur Verfügung. Dieses ist eine spezielle Assembleranweisung, die dafür sorgt, dass der Speicherbus für den nachfolgenden Befehl[3] exklusiv belegt wird – dieser wird dadurch atomar ausgeführt.

Interrupt-Bus

Über den Interrupt-Bus werden Interrupt-Anforderungen, welche zum Beispiel von externen Geräten erzeugt werden, an die Prozessoren weitergeleitet. Da die Prozessoren weder hierarchisch angeordnet sind, noch in einer Master-Slave-Beziehung stehen, kann jeder Prozessor jeden Interrupt entgegennehmen. Bevor auf die Besonderheiten im Multiprozessorsystem eingegangen wird, soll zunächst kurz der Ablauf in Monoprozessorsystemen geschildert werden.

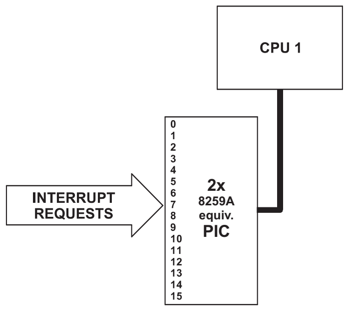


Abbildung 3.2: Interruptsystem im Monoprozessor-PC
(Nach: [Int97b])

Abbildung 3.2 zeigt eine Skizze des Interruptablaufs im Monoprozessor-PC. Dort werden die Interrupts von einem programmierbaren Interrupt-Controller (PIC) entgegengenommen[4]. Der PIC meldet den Interrupt-Wunsch[5], zusammen mit der IRQ-Nummer an die CPU. Diese unterbricht den aktuellen Programmfluss und beginnt die Ausführung einer Interrupt-Behandlungsfunktion. Die Startadresse des Interrupt-Handlers wird mit Hilfe der IRQ-Nummer aus einer Vektortabelle bestimmt [Mes94].

Im Multiprozessorrechner werden die PICs von APICs (**A**dvanced **P**rogrammable **I**nterrupt **C**ontroller) abgelöst. Dabei handelt es sich um ein System aus zwei unterschiedlichen Typen von Interrupt-Controllern: den lokalen APICs und den I/O-APICs (siehe Abbildung 3.3).

Die I/O-APICs nehmen die Interrupts entgegen; dabei kann ein I/O-APIC mit bis zu 24

[3]Das LOCK-Präfix kann nicht vor jeden Befehl gestellt werden. Eine Liste der möglichen Nachfolger und genauere Informationen sind in [Int99a] gegeben.

[4]Es handelt es sich hierbei – historisch bedingt – um zwei PICs, die in einer Master-Slave-Beziehung verschaltet sind. In heutigen Systemen sind sie in der "Southbridge" (z. B. Intel PIIX4 [Int97a]) integriert.

[5]Im Folgenden auch als IRQ (**I**nterrupt-**req**uest) abgekürzt.

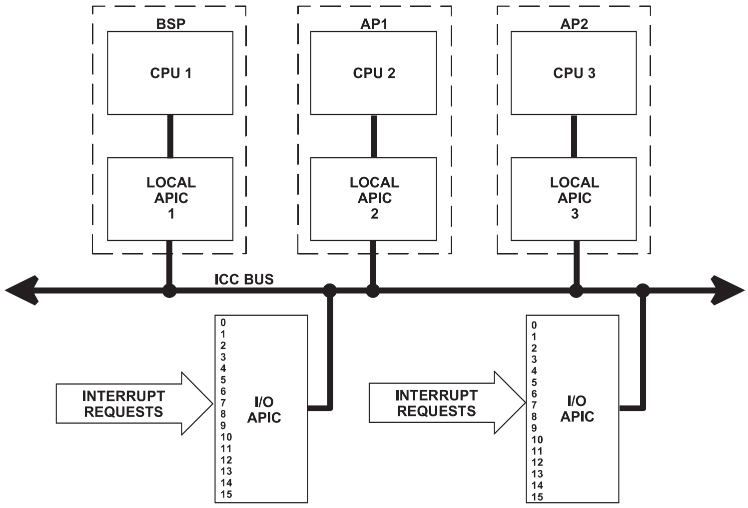


Abbildung 3.3: Interruptsystem in einem Intel-Multiprozessorsystem
(Quelle: [Int97b])

Interruptleitungen verbunden sein.[6] Reicht das nicht aus, weil der MP-Rechner zum Beispiel über viele Geräte-Busse (ISA, PCI, EISA,...) verfügt, so können mehrere I/O-APICs eingesetzt werden. Mindestens einer ist aber in jedem Intel-MP-System vorhanden.

Die lokalen APICs sind für die Benachrichtigung der CPUs verantwortlich. Jeder Prozessor ist deshalb mit einem lokalen APIC verbunden. Die Anzahl der lokalen APICs entspricht somit immer der Anzahl der Prozessoren im System.

An dieser Stelle soll darauf hingewiesen werden, dass es zwei verschiedene Versionen der lokalen APICs gibt. Die erste Version war ein externer Chip (der Intel 82489DX). Er wurde in Systemen eingesetzt, die mit Prozessoren vom Typ Pentium 75 oder älter betrieben wurden. Ab dem Pentium 90 hat Intel die Logik des lokalen APICs in den Prozessor integriert [Int99b]. Die Unterschiede zwischen diesen Versionen werden, soweit sie relevant sind, in späteren Kapiteln noch beschrieben.[7]

Verbunden sind die I/O-APICs und die lokalen-APICs durch den Interrupt-Bus (ICC-Bus). Dabei handelt es sich um einen 3-Draht-Bus, der ausschließlich zur Übermittlung von IRQs genutzt wird und somit den Speicherbus entlastet. Der Weg einer Interrupt-Anforderung läuft somit von einem I/O-APIC über den ICC-Bus zu den lokalen APICs. Diese entscheiden schließlich, ob sie den Interrupt ihrer CPU melden oder nicht. Um zu verhindern, dass zwei I/O-APICs gleichzeitig einen Interrupt melden, wird ein einfacher Arbitrierungsmechanismus mit Round-Robin-Strategie eingesetzt. Genauere Informationen zu dem auf dem APIC-Bus eingesetzten Protokoll finden sich in [Int99b].

[6] Die angegebenen Daten gelten für den I/O-APIC 82093AA von Intel (siehe [Int96]).

[7] Falls nichts anderes erwähnt ist, so beziehen sich alle Angaben über lokale APICs immer auf die integrierte Version, da die externe Version (der 82489DX) nicht mehr hergestellt wird.

3.3.3 Monoprozessor-Kompatibilität

Im vorangegangenen Abschnitt wurde gezeigt, welche Komponenten an der Interruptverteilung in einem Intel-MP-System beteiligt sind. Die Konfiguration dieser Komponenten wird im Kapitel *Interruptsystem* (Abschnitt 4.1) behandelt. Der folgende Teil zeigt zwei verschiedene Arten ein Intel-MP-System aufzubauen, um die Kompatibilität zu einem Monoprozessorsystem zu erhalten.[8]

Beim Aktivieren des Systems sorgt das BIOS dafür, dass grundsätzlich nur ein Prozessor in Betrieb ist, alle anderen Prozessoren werden in einen Wartezustand versetzt. In diesem Zustand bleiben sie solange, bis sie – normalerweise vom Betriebssystem – aktiviert werden. Die CPU, die das System startet, wird als "Bootstrap Processor" – kurz BSP – bezeichnet, alle anderen CPUs werden "Application Processor" – kurz AP – genannt.

Solange nur ein Prozessor aktiv ist, verhält sich das System – aus Sicht der Software – genau wie ein Monoprozessorsystem. Um das zu realisieren, sind für die Verteilung der Interrupts drei verschiedene Interrupt-Modi spezifiziert. Zwei davon (der PIC Mode und der Virtual Wire Mode) beschreiben Möglichkeiten, die Kompatibilität zum Monoprozessor-Rechner beizubehalten, der dritte entspricht dem im vorangegangenen Abschnitt beschriebenen Multiprozessor-Fall.

Bei den beiden ersten Arten werden die Interrupts von einem herkömmlichen PIC – der zusätzlich zu den APICs im System vorhanden ist[9] – entgegengenommen. Der Weg, den ein IRQ danach durchläuft, ist jedoch unterschiedlich:

- PIC Mode (siehe Abbildung 3.4a):
  In diesem Modus werden die Interrupts vom PIC entgegengenommen und direkt an die CPU geleitet. Für den Multiprozessorbetrieb wird dieser Modus deaktiviert (durch das IMCR = **I**nterrupt **M**ode **C**onfiguration **R**egister), sodass der BSP die Interrupts von seinem lokalen APIC erhält.

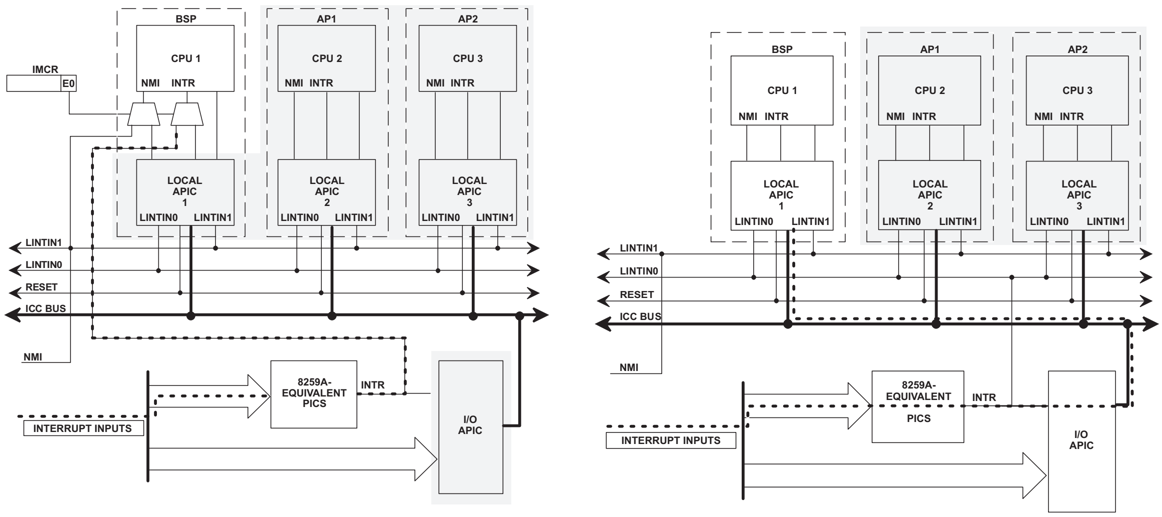
- Virtual Wire Mode (siehe Abbildung 3.4b):
  In diesem Modus werden die Interrupts ebenfalls vom PIC entgegengenommen, jedoch nicht direkt an die CPU geleitet, sondern an den lokalen APIC. Dieser stellt somit ein virtuelles Kabel dar. Um den IRQ zum lokalen APIC zu leiten, kann entweder der ICC-Bus in Verbindung mit einem I/O-APIC – wie in Abbildung 3.4b dargestellt – oder eine dedizierte Leitung (z. B. der LINTIN0 Eingang des lokalen APICs) dienen.

- Symmetric I/O Mode:
  Hier findet die Interruptverteilung wie in Abschnitt 3.3.2 beschrieben statt. Es ist der einzige Modus, in dem Multiprozessor-Betrieb möglich ist.

Welcher der ersten beiden Modi realisiert ist, bleibt dem Hardware-Hersteller überlassen. Wie in Abschnitt 3.3.6 jedoch noch zu sehen ist, muss das Betriebssystem beim Umschalten in den MP-Modus feststellen, welcher der ersten beiden Modi vorliegt.

[8] Dabei wird nur auf die Unterschiede eingegangen, die für ein Betriebssystem relevant sind.

[9] Die Ansteuerung eines APICs unterscheidet sich von der Art, wie ein herkömmlicher PIC angesteuert wird. Aus Gründen der Kompatibilität werden deshalb beide Interrupt-Controller eingesetzt.



(a) PIC Mode

(b) Virtual Wire Mode

(grau hinterlegte Komponenten werden im jeweiligen Modus nicht verwendet)

Abbildung 3.4: Interrupt Modi
(Quelle: [Int97b])

3.3.4 Konfigurationstabelle

Bevor das Betriebssystem in den Mehrprozessor-Modus schalten kann, muss erst festgestellt werden, ob überhaupt ein MP-System vorliegt. Außerdem ist es sinnvoll, Angaben über die Ausstattung des Systems (z. B. Anzahl der CPUs) zu ermitteln.

In einem Intel-MP-System ist hierzu eine Konfigurationstabelle im Speicher vorhanden, die vom BIOS aufgebaut wird und dem Betriebssystem diese Informationen liefert. Die Adresse, unter der die Tabelle abgelegt ist, ist jedoch nicht festgelegt. Stattdessen ist der Ort einer Zeigerstruktur spezifiziert, die neben einigen weiteren Informationen hauptsächlich die Speicherstelle angibt, an der die Konfigurationstabelle beginnt. Abbildung 3.5 zeigt eine Skizze der beiden Strukturen.

Zeigerstruktur

Die Startadresse der Zeigerstruktur ist wiederum nicht eindeutig festgelegt, sondern lediglich auf drei verschiedene Bereiche des Speichers beschränkt (siehe Tabelle 3.1). Um die Struktur zu finden muss das Betriebssystem daher diese Bereiche absuchen. Zur Identifikation ist die Struktur mit einer Signatur (“_MP_”) versehen, welche sonst nicht in diesen Bereichen vorkommen darf.

Die wichtigsten Informationen der Zeigerstruktur sind in Tabelle 3.2 angegeben. Neben der Signatur und der Startadresse der Konfigurationstabelle enthält die Struktur ein Flag, das angibt, welcher der schon in Abschnitt 3.3.3 vorgestellten Interruptmodi realisiert ist. Bei

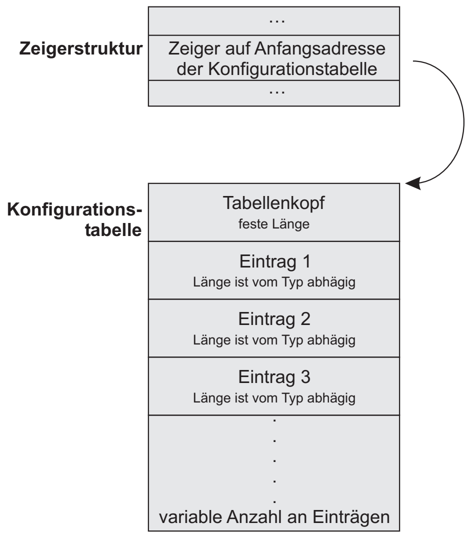


Abbildung 3.5: Zeigerstruktur und Konfigurationstabelle
(Nach: [Int97b])

1.) Erstes KByte der "Extended BIOS Data Area" (EBDA) .
2.) Letztes KByte des Basisspeichers (512 bzw. 640KB).
3.) BIOS ROM Bereich (0x0F0000 - 0x0FFFFF).

Tabelle 3.1: Reihenfolge bei der Suche nach der Zeigerstruktur

Feld	Bemerkung
Signatur	Immer: "_MP_"
Zeiger auf die Konfigurationstabelle	Startadresse oder 0
IMCR-Flag	PIC Mode oder Virtual Wire Mode
Konfigurationstyp	wenn $\neq 0 \Rightarrow$ implizite Konfiguration
Version der MP-Spezifikation	z. Z. 1.1 oder 1.4

Tabelle 3.2: Die wichtigsten Informationen in der Zeigerstruktur

gesetztem Flag ist ein "Interrupt Modus Konfigurations-Register" (IMCR) vorhanden und es liegt somit ein System im PIC-Mode vor; ist das Bit nicht gesetzt, so läuft das System nach dem Bootvorgang im Virtual Wire Mode.

Der Konfigurationstyp gibt an, ob es sich um ein Standard-MP-System handelt. Ist er $\neq 0$, so liegt ein System mit zwei Prozessoren vor, dessen weitere Daten implizit festgelegt sind. Eine Konfigurationstabelle ist in diesem Fall nicht vorhanden, der Zeiger auf die Tabelle hat dementsprechend keine Bedeutung.

Um zu prüfen, ob überhaupt ein MP-System vorliegt, ist es somit ausreichend nach einer Zeigerstruktur zu suchen.[10]

[10]Dies gilt für Intel MP-Spezifikation 1.4 (in früheren Versionen musste zusätzlich noch ein Bit in der Zeiger-

Aufbau der Konfigurationstabelle

Handelt es sich nicht um ein Standardsystem, so existiert eine Konfigurationstabelle, die Informationen über die Ausstattung des Systems enthält. Auf einen Tabellenkopf folgen Einträge für jeden

- Prozessor,
- Bus,
- I/O-APIC und jede
- Interruptleitung.

In Tabelle 3.3 sind die wichtigsten Informationen eines jeden Eintrags aufgeführt:

Tabellenkopf	Signatur
	ID Strings (Hersteller, Produkt)
	Zeiger auf herstellerspezifische Tabelle
	Anzahl an Einträgen
	Basisadresse der lokalen APICs
Prozessor	ID des lokalen APIC
	Version des lokalen APIC
	Verfügbarkeits-Flag
	BSP-Flag
	CPUID (Signatur und Feature-Flags) (siehe [Int00])
Bus	eindeutige ID
	Typ
I/O-APIC	eindeutige ID
	Vefügbarkeits-Flag
	Basisadresse des I/O-APICs
I/O Interrupt Zuordnung	Typ des Interrupts (INT, NMI, SMI oder ExtINT)
	Polarität (high- oder low- aktiv)
	Sensitivität (flanken- oder kantengesteuert)
	ID des Quell-Busses (der Bus von dem der IRQ stammt)
	Nummer des IRQs auf dem Quell-Bus
	ID des Ziel I/O-APIC
	Pin Nr. des Ziel I/O-APIC
lokale Interrupt Zuordnung	wie bei *I/O Interrupt Zuordnung* jedoch für lokale APICs.

Tabelle 3.3: Einträge in der Konfigurationstabelle und ihre wichtigsten Informationen

struktur gesetzt sein).

Der Tabellenkopf enthält neben einigen Identifikationsstrings die Basisadresse, an der die Register der lokalen APICs in den Speicheradressraum eingeblendet werden. Da jeder Prozessor immer nur seinen eigenen lokalen APIC erreichen muss, werden die Register aller lokalen APICs auf die gleichen Adressen abgebildet. Dies ist die einzige Ausnahme, in der die CPUs unter der gleichen Adresse verschiedene Komponenten ansprechen.

Prozessor-Einträge enthalten die ID des lokalen APICs der jeweiligen CPU. Über diese ID werden die Prozessoren identifiziert (siehe Abschnitt 3.3.5). Des Weiteren ist auch die Version des APICs gegeben.

Bus-Einträge weisen jedem Bus im System eine eindeutige Nummer zu, sodass später darauf Bezug genommen werden kann.

I/O-APIC-Einträge geben neben der eindeutigen ID des jeweiligen I/O-APIC auch die Basisadresse an, an der die Register des I/O-APICs in den Adressraum eingeblendet sind.

Interrupt-Einträge sind für jede Interruptleitung vorhanden. Es wird jedoch zwischen zwei verschiedenen Typen unterschieden. Zum einen die Interruptleitungen, die an einen I/O-APIC angeschlossen sind, zum anderen diejenigen, die direkt an einen lokalen APIC angeschlossen sind. Unabhängig vom Typ enthält jeder Eintrag Daten über die Konfiguration des Interrupts (Typ, Polarität, Sensitivität) und über den Verlauf der Leitung (Quelle, Ziel,...). Mit Hilfe dieser Daten muss das Betriebssystem den APIC entsprechend konfigurieren, um den Interrupt korrekt zu interpretieren (siehe dazu Abschnitt 4.1).

Neben den genannten Einträgen wurden mit der Version 1.4 der Intel-MP-Spezifikation [Int97b] noch einige erweiterte Einträge eingeführt, zum Beispiel um die hierarchischen Beziehungen der Busse zu definieren. Auf diese erweiterten Einträge soll hier jedoch nicht eingegangen werden.

3.3.5 Identifikation eines Prozessors

Um verschiedene Prozessoren anzusprechen, muss man sie unterscheiden können. Dazu dient die ID ihres lokalen APICs. Es gibt nun zwei grundsätzlich verschiedene Möglichkeiten einen Prozessor zu adressieren:

- physikalische Adressierung:
  Hierzu hat jeder lokale APIC ein 4-Bit breites ID Register[11], welches binär codiert verwendet wird. Daraus ergibt sich eine maximale Anzahl von 15 Prozessoren im System (der ID-Wert 15 wird für einen Broadcast genutzt).

[11] Die Angaben beziehen sich auf die integrierte Version des lokalen APIC.
Die externe Version (der 82489DX) hat breitere ID-Register.

- logische Adressierung
  Hierfür steht – unabhängig vom 4-Bit ID Register der physikalischen Adressierung – ein 8-Bit breites ID-Register zur Verfügung. Es gibt zwei Modelle, das Register zu nutzen:
  - flaches Modell
    Hier werden alle 8-Bit als ID einer CPU genutzt. Im Gegensatz zur physikalischen Adressierung wird die ID jedoch unär codiert. Das heißt, dass je ein Bit für einen Prozessor steht. Die maximale Anzahl an Prozessoren ist somit auf acht begrenzt.
  - Cluster Modell
    Bei dieser Variante der logischen Adressierung wird die CPU ID nur in den 4 niederwertigen Bit des ID Registers gespeichert. Die anderen 4 Bit dienen zur Bildung von Gruppen (Cluster). Da die Cluster ID binär gespeichert wird, die CPU ID unär, können somit insgesamt (15 Cluster $\times$ 4 CPUs =) 60 CPUs adressiert werden.
    Da an jedem APIC-Bus jedoch höchstens 15 Geräte angeschlossen werden dürfen,[12] kann das Cluster Modell wiederum in zwei Varianten aufgeteilt werden:
    - flaches Cluster Modell
      Bei diesem Modell ist nur ein APIC-Bus (=ICC-Bus) im System vorhanden. Die maximale CPU-Anzahl ist somit auf 15 begrenzt.
    - hierarchisches Cluster Modell
      Es gibt auch die Möglichkeit mehrere APIC-Busse zu koppeln. Hierbei übernehmen so genannte Cluster-Manager die Verteilung der Nachrichten auf die einzelnen APIC-Busse. Eine Gesamtanzahl von 60 Prozessoren kann somit angesprochen werden.

Die Vor- und Nachteile der verschiedenen Adressierungsarten liegen vor allem in ihrer Flexibilität und in der Anzahl maximal ansprechbarer Prozessoren. Mit der physikalischen Adressierung kann man entweder alle CPUs oder einen bestimmten Prozessor adressieren – eine Kommunikation mit einer Gruppe ist nicht möglich. Diesen Nachteil hebt das flache Modell der logischen Adressierung auf, jedoch mit der Folge, dass nur noch acht Prozessoren ansprechbar sind. Das Cluster-Modell wiederum ist als Kompromiss zwischen den zwei anderen Lösungen anzusehen.[13]

Beide ID-Register sind vom Betriebssystem beschreibbar. Daraus folgt, dass die ID jedes Prozessors frei geändert werden kann. Allerdings muss darauf geachtet werden, dass jede CPU eine eindeutige Nummer erhält, da sonst keine korrekte Kommunikation mehr möglich ist.

3.3.6 Aktivierung eines Prozessors

In Abschnitt 3.3.3 wurde schon beschrieben, dass nach dem Starten des Systems nur ein Prozessor aktiv ist, während die anderen in einem Wartezustand sind. Deshalb müssen zusätzliche Prozessoren erst aktiviert werden, bevor sie genutzt werden können.

[12] Dafür ist der Arbitrierungs-Mechanismus auf dem APIC-Bus verantwortlich. Die Bus-Zuteilung wird durch eine Round-Robin-Strategie mit einer 4-Bit breiten (Arbitrierungs-)ID durchgeführt (siehe [Int99b]).

[13] Außerdem ist es die einzige Adressierungsmöglichkeit in Systemen mit mehr als 15 CPUs.

Um einen Prozessor zu aktivieren, muss ihm von einem schon aktiven Prozessor (meist dem BSP) eine bestimmte Folge von Interrupts – so genannte Interprozessorinterrupts (IPI) – geschickt werden. Daraufhin führt die angesprochene CPU einen Warmstart durch und steht dem System zur Verfügung. In Abbildung 3.6 ist das Vorgehen in Pseudocode wiedergegeben.

```
/* symmetric I/O- Modus aktivieren */
if (PIC mode) then
    set IMCR to accept IRQs from local APIC
end if
activate local APIC

/* Start-Code festlegen */
set BIOS shut down code to warm reset(0x0A)
set warm reset vector to the AP startup code

/* IPIs senden */
send INIT IPI to AP
if (local APIC = integrated) then
    send STARTUP IPI to AP
    /* evtl. wiederhohlen */
end if

/* Funktions- Prüfung */
/* der aktivierte Prozessor sollte dazu nach
   seiner Aktivierung ein Flag setzen */
if (Flag = TRUE) then
    OK
else
    ERROR
end if
```

Abbildung 3.6: Pseudocode zum Aktivieren eines Prozessors

Aktivierungssequenz

Die Aktivierungssequenz kann man in vier Phasen einteilen:[14]

1. Aktivieren des Interruptsystems
   Zuerst muss das Interruptsystem in den Multiprozessor-Modus gebracht werden, um Interrupts verschicken zu können. Hierbei ist zu unterscheiden, ob ein System mit PIC Modus oder Virtual Wire Modus vorliegt (siehe Abschnitt 3.3.3 und 3.3.4). Falls der Virtual Wire Modus realisiert ist, so muss lediglich der lokale APIC aktiviert werden. Wenn es sich um ein System mit PIC Modus handelt, so muss zusätzlich das IMCR (**I**nterrupt **M**ode **C**onfiguration **R**egister) so eingestellt werden, dass zukünftige Interrupts durch den lokalen APIC des BSP empfangen werden und nicht mehr direkt zum

[14] Es wird davon ausgegangen, das der BSP die Aktivierung durchführt.

Prozessor geleitet werden. Erst dadurch kann der BSP Interrupts von anderen CPUs empfangen.

2. Festlegen des Startcodes
   Da der Prozessor einen Warmstart durchführen soll, ist im BIOS festzulegen, an welcher Adresse die Ausführung beginnen soll. Außerdem ist einzustellen, dass ein Warmstart gewünscht ist, da ansonsten das gesamte System neu gestartet wird,[15] nicht nur der angesprochene Prozessor.

3. Senden von IPIs zum Aktivieren
   Nun können die Nachrichten zum AP ("Application Prozessor") gesendet werden, um ihn zu starten. Sein lokaler APIC muss dabei nicht explizit aktiviert werden, da die beiden verwendeten Interrupts immer empfangen und verarbeitet werden.
   Zuerst wird ein INIT IPI gesendet. Dieser Interrupt veranlasst den Warmstart des Prozessors. Falls die integrierte Version des lokalen APICs vorliegt, so muss anschließend ein STARTUP IPI gesendet werden. Mit diesem Interrupt wird erneut eine Startadresse festgelegt (mitgeschickt). Sie entspricht im Normalfall der Warmstartadresse. Der Prozessor beginnt anschließend die Ausführung an der angegebenen Adresse.

4. Funktionsprüfung
   Anschließend muss sichergestellt werden, dass der aktivierte Prozessor auch korrekt arbeitet. Dazu sollte der Startcode der neu aktivierten Prozessoren zuerst ein Bit im gemeinsamen Speicher setzen, welches der BSP abfragt (pollt) und somit feststellen kann, ob die Aktivierung erfolgreich war oder nicht.

Startcode

Beim Festlegen der Startadresse ist darauf zu achten, dass der Startcode im ersten MByte des Speichers liegen muss. Diese Einschränkung kommt daher, dass sich die Prozessoren nach dem Start im "Real-Mode" (siehe [Mes94]) befinden. In dieser Betriebsart kann der Prozessor jedoch nur den unteren Speicherbereich (Adressen bis 1 MByte) ansprechen.

Üblicherweise führt ein Startcode folgende Arbeiten durch:

- Kontroll-Bit setzen (zur Funktionskontrolle).
- Umschalten in den "Protected-Mode" (siehe [Mes94] und [WM95]).
- Installieren einer Interrupt-Tabelle (siehe [Int99b, Mes94]).
- Betriebssystemspezifische Aufgaben (z. B. Anmeldung beim Scheduler).

[15] Der gesamte BIOS Initialisierungs-Code (POST) würde ausgeführt werden.

3.4 Implementierung

Sequentielle Aktivierung der Prozessoren

Die Implementierung im Rahmen von JX erfordert das Erkennen und Aktivieren der Multiprozessorumgebung. Der Code für diese Aufgaben ist in den Dateien `smp_detect.c` und `smp_activate.c` abgelegt.

Die Realisierung ist an den Linux-Kernel 2.2.12 angelehnt und geht sequentiell vor:

- Zuerst wird festgestellt, ob es sich um ein MP-System handelt (`smp_detect.c`). Wurde eine MP-Zeigerstruktur gefunden, wird anschließend die MP-Konfigurationstabelle gelesen, um die Informationen für eine spätere Verwendung zu speichern.[16] Handelt es sich um eine Standard-Konfiguration, so werden die – implizit gegebenen – Daten ebenfalls bereitgestellt, als ob sie aus einer Konfigurationstabelle stammten.
- Anschließend erfolgt das Starten der CPUs, wie in Pseudocode (Abbildung 3.6) dargestellt (`smp_activate.c`). Dazu wird zuerst der Startcode in das erste Megabyte des Hauptspeichers kopiert. Die Adresse kann dabei beliebig gewählt werden, da der gesamte untere Speicherbereich unbenutzt ist. Das liegt daran, dass der Code von JX (Kern und Anwendungsmodul) durch einen "Bootloader" [FB96] in den Speicher oberhalb des ersten Megabytes geladen wurde.
- Der Startcode selbst führt die in Abschnitt 3.3.6 aufgelisteten Aufgaben durch. Anschließend aktiviert er den Idle-Thread.
  Wird später ein Scheduler für diesen Prozessor registriert, so erhält dieser mit Hilfe eines IPIs die Kontrolle über die CPU und kann den Timer-Interrupt aktivieren.

Alternative Realisierungen

Eine andere Art der Realisierung wäre ein paralleler Ansatz: das gleichzeitige Aktivieren aller vorhandenen Prozessoren. Dadurch könnte der Startvorgang beschleunigt werden. Jedoch wird dadurch die Aktivierungssequenz komplexer und unübersichtlicher. Außerdem stand zur Implementierung nur ein Dual-Prozessor-System zur Verfügung. Für ein System mit nur zwei CPUs ist jedoch kein Unterschied erkennbar, ob alle Prozessoren gleichzeitig oder sequentiell aktiviert werden.[17] Ein Testlauf der parallelen Aktivierung hätte somit ohnehin nicht durchgeführt werden können.

Eine weitere Alternative wäre die Realisierung in Java gewesen. Es wäre somit möglich die vorhandene Klassen zum Lesen der PCI-Informationstabellen auch zur Verarbeitung der MP-Konfigurationstabelle zu verwenden. Somit würden die Daten an einer zentralen Stelle gesammelt. Außerdem könnte die Aktivierung der zusätzlichen Prozessoren dann an beliebiger Stelle – insbesondere auch zu einem späteren Zeitpunkt – stattfinden.

[16] Neben der Festlegung von Anzahl und ID der CPUs dienen die Informationen hauptsächlich der Interrupt-Konfiguration (siehe Abschnitt 4.1).

[17] Bei zwei Prozessoren muss nur einer aktiviert werden, der andere ist der BSP (**B**oot**s**trap **P**rozessor)

Allerdings stellt sich die Frage, wie sinnvoll diese Möglichkeiten sind. Ein zentrale Speicherung der Daten ist nicht unbedingt von Vorteil, wenn die Daten – so wie in diesem Fall – nur von einem Konsumenten benötigt werden.[18] Durch eine flexiblere Gestaltung der Aktivierungssequenz steigt darüberhinaus die Komplexität, wodurch der Code anfälliger für Fehler wird. Außerdem gibt es so gut wie keinen Fall, in dem die Aktivierungssequenz verändert werden müsste.

Verwaltung der Prozessoren

Sind die CPUs gestartet und verfügbar, so wird für jeden Prozessor ein CPU-Objekt erzeugt. Durch dieses Objekt wird der Prozessor repräsentiert. Will man beispielsweise einen Thread auf einem bestimmten Prozessor ausführen, so wird das entsprechende CPU-Objekt benötigt (siehe Anhang A.3).

Wie die meisten Ressourcen in JX werden auch die CPUs, welche eine Domain verwenden darf, vorgegeben. Das heißt, dass beim Erzeugen einer neuen Domain neben der Größe des Speichers auch die CPU-Objekte mitgegeben werden müssen, die von der neue Domain verwendet werden dürfen. Da ein dynamisches Nachfordern der CPUs nicht vorgesehen ist, kann sichergestellt werden, dass eine Domain niemals mehr Prozessoren verwendet als ihre erzeugende Domain.

3.5 Zusammenfassung

Intel-Multiprozessorsysteme sind eine Erweiterung der PC/AT kompatiblen Monoprozessorsysteme. Software, die auf einem solchen Monoprozessorsystem läuft, kann ohne Veränderung auf einem Intel-MP-System eingesetzt werden. Um jedoch die Vorteile eines MP-Systems nutzen zu können, muss zumindest das Betriebssystem auf die erweiterte Hardware eingehen und sie entsprechend unterstützen. Informationen über die vorhandenen Komponenten gibt eine vom BIOS bereitgestellte Konfigurationstabelle. Das Aktivieren zusätzlicher Prozessoren erfolgt über spezielle Interprozessorinterrupts, die für einen Warmstart der neuen Prozessoren sorgen.

[18] In der Implementierung zu dieser Arbeit wurden die Daten nur zur Konfiguration der Interrupt-Controller genutzt (siehe Abschnitt 4.1).

4 Interruptsystem

Das grundlegende Interruptsystem von JX, so wie es vor Beginn dieser Arbeit vorlag (siehe Abschnitt 2.3), wurde nicht verändert. Jedoch mussten einige Anpassungen vorgenommen werden, um auf die Anforderungen eines Multiprozessorsystems eingehen zu können.

Der folgende Abschnitt beschreibt die Konfiguration der I/O-APICs. Anschließend werden die lokalen APICs genauer betrachtet. Im letzten Teil des Kapitels wird auf einige Besonderheiten der Implementierung eingegangen.

4.1 Konfiguration der I/O-APICs

Für die flexible Verwaltung der Interrupts im Multiprozessorsystem sind die I/O-APICs verantwortlich. Wie in Abschnitt 3.3.2 schon erwähnt wurde, nehmen sie die Interrupts entgegen und leiten sie auf den Interrupt-Bus.

Im Gegensatz zum Monoprozessor-Rechner ist im MP-System nicht eindeutig definiert, welche Interruptleitung mit welcher (Interrupt-)Vektornummer verknüpft ist. Das wäre aufgrund der vielseitigen Ausstattungsmöglichkeiten eines Intel-MP-Systems auch gar nicht sinnvoll. Stattdessen sind Informationen über die Interruptleitungen in der Konfigurationstabelle hinterlegt (vgl. Abschnitt 3.3.4). Das Betriebssystem kann mit Hilfe dieser Informationen entscheiden, welche Interruptquelle mit welchem Interruptvektor verbunden werden soll.

Der I/O-APIC hat hierzu eine programmierbare Umsetzungstabelle, welche für jede Interruptleitung (jeden Pin) Informationen enthält, wie eine Pegelveränderung zu interpretieren ist. Diese Tabelle kann bis zu 240 Einträge speichern, übliche I/O-APICs haben jedoch nur 24 Einträge[1][Int96]. Tabelle 4.1 zeigt den Aufbau eines Eintrags:

Interrupt Vektor: Hier legt man fest, welche Vektornummer der Interruptleitung zugeordnet werden soll. Werte zwischen 0 und 15 sind für den Prozessor reserviert, alle folgenden Werte (bis 255) sind frei verfügbar. Da die Interruptvektortabelle des Prozessors ebenfalls frei programmierbar ist, ist die Wahl der Nummer auf den ersten Blick irrelevant. Jedoch spiegelt sie die Priorität des IRQs wieder. Je 16 aufeinanderfolgende Werte haben die

[1] Der Intel 82093AA I/O-APIC hat 24 Interrupteingänge (INTIN[0:23]) und deshalb auch 24 Einträge in der Umsetzungstabelle.

Feld	Beschreibung	Breite (Bit)
Interrupt Vektor	Vektornummer (16 bis 255)	8
Destination Mode	logische oder physikalische Prozessoradressierung	1
Destination Field	ID des Zielprozessors	8
Delivery Mode	legt die Reaktion der lokalen APICs auf einen Interrupt fest	3
Pin Polarity	0 = high aktiv, 1 = low aktiv	1
Trigger Mode	0 = kantensensitiv, 1 = flankensensitiv	1
Interrupt Mask	Maskierungs-Bit	1

(einige unbedeutendere Felder wurden aus Gründen der Übersichtlichkeit nicht mit angegeben)

Tabelle 4.1: Die Umsetzungstabelle eines I/O-APICs
(Nach: [Int96])

gleiche Priorität.[2] Einer Prioritätsstufe sollte jedoch nach Möglichkeit maximal zwei Interrupts zugeordnet werden, da ansonsten IRQs verloren gehen können.[3]

Destination Mode: Durch dieses Feld wird festgelegt, ob die Zieladresse logisch oder physikalisch angegeben wird (die Adressierungsarten wurden in Abschnitt 3.3.5 beschrieben)

Destination Field: Hier gibt man die Adresse des oder der Prozessoren an, die über diesen IRQ informiert werden sollen. Je nach Zielmodus muss es sich dabei um eine logische oder physikalische Adresse handeln.

Delivery Mode: Der Liefermodus legt fest, wie die adressierten lokalen APICs reagieren sollen. Es gibt dabei folgende Möglichkeiten:[4]

- **fixed:** der Interrupt wird an alle adressierten Prozessoren gemeldet.
- **lowest priority:** unter allen adressierten Prozessoren wird der Interrupt nur der CPU gemeldet, deren Prioritätsregister den niedrigsten Wert enthält. Das Prioritätsregister kann vom Betriebssystem beschrieben werden, um der Hardware mitzuteilen, wie wichtig der Code ist, der gerade von einem Prozessor ausgeführt wird. Somit unterbricht ein Interrupt in diesem Modus immer die Ausführung des Codes mit geringster Priorität.
- **SMI:** der IRQ wird als System Management Interrupt gemeldet.
- **NMI:** der IRQ wird an den NMI-Eingang (**n**on-**m**askabel **I**nterrupt) aller angegebenen Prozessoren gemeldet.

[2] Die höherwertigen 4 Bit der Vektornummer stehen somit für die Priorität: 1 bedeutet niedrigste Priorität, 15 steht für die höchste Priorität.

[3] Der lokale APIC hat für jede Prioritätsstufe ein "**In-S**ervice-**R**egister" (ISR) und ein **I**nterrupt-**R**equest-**R**egister" (IRR) in denen gespeichert wird, welcher Interrupt gerade vom Prozessor verarbeitet wird bzw. welcher IRQ noch vorliegt. Kommen drei Interrupts der selben Prioritätsstufe zur gleichen Zeit an, so geht auf jeden Fall einer verloren. Zwei IRQs können noch gespeichert werden, wenn der erst sofort bearbeitet werden kann.

[4] Je nach Modus sind einige der anderen Felder wirkungslos.

- **INIT:** der IRQ wird an den INIT-Eingang[5] aller adressierten Prozessoren gemeldet.
- **ExtINT:** die Vektornummer des IRQs wird von einem externen Interrupt-Controller (PIC) bereitgestellt.

Polarity, Trigger Mode: legt die Eigenschaften des Eingangssignals fest.

Interrupt Mask: mit Hilfe diese Bits kann man die Weiterleitung eines Interrupts deaktivieren.

4.2 Lokale APICs

4.2.1 Konfiguration

Ein lokaler APIC hat neben den Pins für den ICC-Bus auch noch einige Eingänge für lokale Interrupts.[6] Die Reaktion auf Pegelveränderungen an diesen Pins lässt sich ähnlich konfigurieren wie bei den I/O-APICs. An dieser Stelle soll deshalb auch nicht näher darauf eingegangen werden, genaue Informationen hierzu finden sich in [Int99b].

4.2.2 Interprozessorinterrupts

Mit Hilfe der lokalen APICs ist es möglich, so genannte Interprozessorinterrupts (IPI) zu verschicken. Dabei handelt es sich um Interrupts, die nicht von externen Geräten ausgelöst werden, sondern die von einem Prozessor zu einem anderen geschickt werden.

Verwendung finden IPIs zum Beispiel – wie in Abschnitt 3.3.6 schon erwähnt – bei der Aktivierung anderer Prozessoren. Allerdings können sie auch vom Betriebssystem zur Koordination der Prozessoren genutzt werden. So könnte man den Zugriff auf eine Datenstruktur auf eine CPU beschränken, um den gegenseitigen Ausschluss leichter zu gewährleisten. Will ein anderer Prozessor die Daten verändern, wendet er sich mit Hilfe eines IPIs an den befugten Prozessor, der daraufhin die Änderung vornimmt.

Als Ziel eines IPIs kommen folgende Möglichkeiten in Betracht:

- ein einzelner anderer Prozessor,
- der Sender selbst (man spricht dann von einem “Self-Interrupt”),
- alle anderen Prozessoren außer dem Sender,
- alle CPUs inklusive dem Sender oder
- eine Gruppe von Prozessoren, falls eine Adressierung nach dem Cluster-Modell (Abschnitt 3.3.5) erfolgt.

[5] Zur Wirkungsweise des INIT Signals siehe [Int98].
[6] Erkennbar in Abbildung 3.4.

4.2.3 Lokaler Timer

Neben der Interruptverwaltung stellt jeder lokale APIC noch einen 32 Bit-Zähler – "local Timer" genannt – zur Verfügung. Dieser wird bei jedem Systembustakt erhöht und kann dem jeweils zugeordneten Prozessor in programmierbaren Abständen Interrupts zustellen. Die Vektornummer ist dabei wieder frei konfigurierbar.

Einsatzgebiete für diesen Zähler sind dort, wo im Monoprozessorsystem der Intervall-Timer [Mes98] benutzt wird. Würde man diesen auch in Multiprozessorsystemen verwenden, so würden alle CPUs gleichzeitig benachrichtigt. Mit den lokalen Zählern hat man den Vorteil, dass man für jeden Prozessor unterschiedlich lange Intervalle definieren kann. Als Nachteil kann sich jedoch auswirken, dass die Geschwindigkeit des Zählers vom Systemtakt abhängt und nicht – wie die des Intervall-Timers – durch einen separaten, standardisierten Takt erzeugt wird.

Im Rahmen von JX werden die lokalen Timer eingesetzt, um für jeden Prozessor eine unabhängige Zeitscheibe für den Scheduler zu realisieren (siehe hierzu Abschnitt 5.6).

4.3 Implementierung

Konfiguration des I/O-APICs und Interruptsteuerung

Um in JX die Möglichkeit zu nutzen, den Zielprozessor eines Interrupts frei wählen zu können, werden die IRQs – falls ein MP-System vorliegt – über die I/O-APICs verwaltet. Dazu wird die Umsetzungstabelle (in den IO-APICs), wie in Abschnitt 4.1 beschrieben, nach den Daten aus der MP-Konfigurationstabelle konfiguriert (in io_apic.c). Die Vektornummern werden dabei gleichmäßig verteilt. Nummern größer als 240 bleiben jedoch für das Betriebssystem reserviert und werden nicht an Hardware-Interrupts vergeben.

Falls einige Interruptleitungen nicht mit dem I/O-APIC verbunden sind,[7] so werden diese weiterhin vom PIC verwaltet. Demzufolge müssen die Kommandos zur Deaktivierung und Bestätigung eines Interrupts an den entsprechenden Interrupt-Controller geschickt werden. Um das flexibel zu realisieren, werden die Befehle über eine Tabelle mit Funktionenzeiger geleitet. Ein Eintrag dieser Tabelle zeigt entweder auf Funktionen die den PIC ansprechen oder auf Funktionen für die Steuerung eines I/O-APICs.

Als Konsequenz daraus muss jeder Befehl zur Interruptsteuerung erst den Umweg über die Tabelle nehmen. Der daraus resultierende Geschwindigkeitsverlust fällt bei der Deaktivierung bzw. Aktivierung eines IRQs wenig ins Gewicht, da diese Befehle selten verwendet werden. Für die Bestätigung eines IRQs bedeutet es jedoch eine geringfügig verlängerte Latenzzeit.

[7] Bei manchen Standard-Konfigurationen ist die Interruptleitung des Intervall-Timers nicht mit dem I/O-APIC verbunden.

Individueller Interrupthandler pro Prozessor

Die Installation eines Interrupthandlers wurde, gegenüber dem zu Beginn dieser Arbeit vorhandenen System, leicht erweitert. Im MP-System kann für jeden IRQ nicht mehr nur ein Handler installiert werden, sondern jeder Prozessor kann einen eigenen Handler bereitstellten. Bei der Installation eines Interrupthandlers wird deshalb anhand der CPU-ID der aktuelle Prozessor bestimmt. Für diese CPU wird dann der IRQ-Handler (bzw. dessen Startadresse) gespeichert.

Um nach einem Interrupt den entsprechenden Code auszuführen, könnte man für jeden Prozessor eine eigene *Interrupt Descriptor Table* (IDT) [Mes94] anlegen. In dieser Tabelle ist für jeden Interruptvektor festgelegt, an welcher Speicherstelle die Bearbeitung fortgesetzt werden soll sobald der jeweilige IRQ auftritt.

Die aktuelle Implementierung verwendet für jeden Prozessor jedoch dieselbe Tabelle. Bei einem Interrupt ermittelt der ausgeführte Code dann die Prozessor-ID und entscheidet prozessorabhängig, welcher Interrupt-Handler auszuführen ist.

Vorteil einer prozessorabhängigen IDT wäre, dass der aktivierte Code die Prozessor ID nicht mehr feststellen müsste, da sie implizit gegeben ist. Eine Veränderung dahingehend würde allerdings größere Änderungen des bestehenden Systems nach sich ziehen. Deshalb wurde darauf verzichtet, zumal der zu erwartende Geschwindigkeitsgewinn nur gering wäre.

Interprozessor-Interrupts

Um die Kommunikation mittels IPIs zu nutzen, steht im JX-Kern eine C-Funktion zur Verfügung, mit deren Hilfe man einen anderen Prozessor veranlassen kann, eine angegebene Funktion auszuführen:

```
int smp_call_function (int dest, int (*func) (void *info),
                                   void *info, int wait, int *result)
```

dest gibt den Zielprozessor an; neben einem einzelnen Prozessor ist auch die Adressierung aller möglich.

func ist ein Zeiger auf die Funktion, die ausgeführt werden soll.

info ist der Parameter welcher der Funktion mitgegeben wird.

wait gibt an, ob auf die erfolgreiche Ausführung der Funktion gewartet werden soll (synchrone Ausführung).

result ist ein Zeiger auf eine Variable die das Ergebnis der Funktion aufnehmen soll.

Die Arbeitsweise der Funktion smp_call_function ist wie folgt:

- Zuerst werden alle Informationen, die für den Aufruf der Funktion auf einer anderen CPU benötigt werden, in einer Struktur zusammengefasst.
- Anschließend wird ein globaler Zeiger auf die Parameterstruktur gesetzt. Durch eine (atomare) "Compare-and-Swap" Operation wird dabei gewährleistet, dass der Zeiger vorher unbenutzt (NULL) war. Falls das nicht der Fall war, so ruft gerade ein anderer Prozessor eine Funktion mit Hilfe dieses Mechanismus auf. Die Funktion smp_call_function kehrt daher mit einer entsprechenden Meldung zurück und der aktuelle Prozessor kann den Aufruf erneut versuchen.[8]
- War der Zeiger unbenutzt, so sind die Vorbereitungen abgeschlossen und ein spezieller IPI wird ausgelöst, der für diesen Zweck eingerichtet wurde.
- Daraufhin wird bei allen adressierten Prozessoren die Behandlungsfunktion für diesen Interrupt aktiviert. Zu beachten ist, dass sie direkt in dem unterbrochenen Kontext ausgeführt wird. Es findet keine Umschaltung zu einem Interruptthread statt, wie bei normalen Interrupts.
- Im Interrupt-Handler werden die Parameter in einen eigenen Speicherbereich kopiert. Anschließend wird dem Aufrufer mitgeteilt, dass die Ausführung der Funktion beginnt. Dazu ist ein spezielles Feld in der Struktur vorgesehen, welches jeder Prozessor inkrementiert.
- Hat der Aufrufer eine Rückmeldung von allen angesprochenen CPUs erhalten, so kann er sicher sein, dass die Parameter jeweils in einen eigenen Speicherbereich kopiert wurden. Der globale Zeiger kann nun wieder auf NULL zurückgesetzt werden.

4.4 Zusammenfassung

Durch entsprechende Konfiguration der I/O-APICs kann man den Zielprozessor eines Interrupts frei wählen. Des Weiteren kann man für jeden Prozessor einen eigenen Interrupt-Handler hinterlegen.

Mit Hilfe der lokalen APICs können Interrupts zwischen den Prozessoren verschickt werden. Dadurch kann eine asynchrone Kommunikation zwischen den CPUs aufgebaut werden. Im JX-Kern wird die Möglichkeit, eine Funktion auf einem anderen Prozessor auszuführen, mit Hilfe dieser Interprozessor-Interrupts realisiert.

[8] Falls die Interrupts gesperrt sind, so hat der Prozessor dadurch die Gelegenheit sie kurzzeitig freizugeben. Sollte er nämlich Ziel des zur Zeit laufenden Aufrufs sein, so muss dieser zuerst abgeschlossen werden. Mit deaktivierten Interrupts kann auf den IPI eines anderen Prozessors jedoch nicht reagiert werden; eine "Deadlock" würde entstehen.

5 Scheduler

In diesem Kapitel wird ein multiprozessorgeeignetes Schedulingkonzept für JX vorgestellt. Nach einer kurzen Einführung folgt die Definition der Anforderungen. Anschließend wird das zugrundeliegende Konzept erläutert. Daraufhin wird ein tieferer Einblick in die Interaktionsmechanismen und deren Realisierung gegeben. Abschließend wird die Zeitscheibenverwaltung vorgestellt, die das Konzept abrundet.

5.1 Überblick

In Multiprozessorsystemen haben Scheduler nicht nur die Aufgabe, die Aktivitätsträger zu verwalten, sondern sie müssen auch die Prozessorzuteilung vornehmen. Dabei können viele Faktoren eine Rolle spielen. In einem inhomogenen MP-System beispielsweise müssen die Unterschiede zwischen den Prozessoren berücksichtigt werden, da nicht jede CPU jeden Auftrag erfüllen kann. In homogenen MP-Systemen steht stattdessen die Lastverteilung oder die Prozessoraffinität[1] eines Threads im Vordergrund.

Bei der Konzeption von Schedulern für Multiprozessorsysteme sind drei grundsätzliche Vorgehensweisen zu unterscheiden:

- Eine Warteschlange pro Prozessor
  Dieser Ansatz hat den Vorteil, dass keine Synchronisation der Warteschlangen vorgenommen werden muss. Allerdings führt dieses Konzept leicht zu einer ungleichen Auslastung der Prozessoren, wenn beispielsweise die Aufträge gleichmäßig verteilt werden, die Bearbeitungsdauer der Aufträge jedoch sehr unterschiedlich ist. Man muss deshalb eine Möglichkeit vorsehen, Aufträge zwischen den Warteschlangen der einzelnen Prozessoren zu verschieben.

- Eine gemeinsame Warteschlange für alle Prozessoren
  Der Effekt der ungerechten Auslastung kann hierbei nicht auftreten. Jedoch müssen hier die Zugriffe auf die Warteschlange koordiniert werden. Dies hat im schlechtesten Fall zur Folge, dass sich die Prozessoren andauernd an der Warteschlange synchronisieren müssen und somit nicht effizient arbeiten können.

[1] Unter der Berücksichtigung der Prozessoraffinität versteht man, dass einem Thread immer der Prozessor zugeteilt wird, auf dem er zuletzt aktiv war. Die Wahrscheinlichkeit, dass sich noch Daten des Threads im Cache befinden ist bei diesem Prozessor am größten.

- Master/Slave Architektur
  Hier wird ein Prozessor für das Scheduling herangezogen, der alle anderen mit Aufträgen versorgt. Das Koordinationsproblem ist dabei nach dem Prinzip eines Monitors gelöst, da alle Zugriffe auf die Warteschlange vom Master-Prozessor durchgeführt werden.
  Der Nachteil dieses asymmetrischen Verfahrens ist, dass dieses Modell nur schlecht skaliert, da der Master schnell zum Engpass werden kann. Umgekehrt kann aber auch der Master "leer stehen", während alle Slave-Prozessoren arbeiten.

Aus den angesprochenen Ansätzen können auch beliebige Kombinationen erzeugt werden. So kann man mehrstufige Scheduler entwickeln, die genau auf die Hardware eines Systems zugeschnitten sind. Für eine NUMA Architektur kann es beispielsweise sinnvoll sein, jedem Prozessor eine eigene Warteschlange zuzuteilen, die in seinem Speicher abgelegt werden kann. Zur Lastverteilung könnte darüberhinaus noch eine zusätzliche globale Warteschlange eingeführt werden. Hat ein Prozessor zu viele Aufträge (Threads), so gibt er welche ab; hat er zu wenige, so entnimmt er Aufträge aus der globalen Warteschlange.

5.2 Ausgangssituation

Zu Beginn dieser Arbeit wurde die Verwaltung der Threads in JX mit Hilfe einer globalen Warteschlange realisiert, die alle lauffähigen Threads des Systems enthielt (siehe Abbildung 5.1). Zur Aktivierung des Schedulers diente der Interrupt des Intervall-Timers. Der aktivierte Scheduler führte daraufhin einen Threadwechsel durch. Dabei wurde eine Round-Robin-Strategie eingesetzt, um den nächsten Aktivitätsträger auszuwählen. Da die Schedulingstrategie im Kern realisiert wurde, konnte kein Einfluss darauf genommen werden. Die Schnittstellen im Kern waren beschränkt auf die Möglichkeiten Threads in die Warteschlange zu stellen und den nächsten (lauffähigen) Thread zu entnehmen.

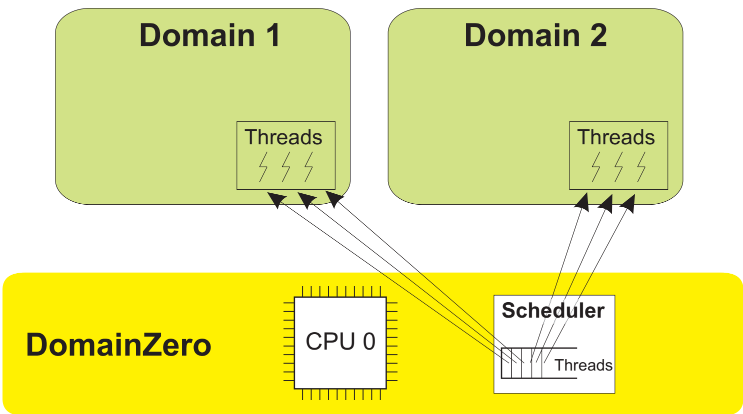


Abbildung 5.1: Ausgangssituation: globaler Scheduler

Da alle Threads gleichberechtigt behandelt wurden – egal zu welcher Domain sie gehörten – wurde die Rechenzeit, bezüglich der Domains, ungerecht aufgeteilt. Einer Domain, die mehr

Threads hatte, wurde der Prozessor öfter zugeteilt als einer Domain, die weniger Threads hatte.

Außerdem war das Scheduling für die Domains transparent. Das heißt, sie hatten keinen Einfluss darauf, wann ihr Thread unterbrochen und welcher Aktivitätsträger als nächstes aktiviert wurde. Sie konnten auch nicht feststellen, ob sie überhaupt unterbrochen wurden.

5.3 Anforderungen und Zielsetzung

Für ein multiprozessorfähiges Betriebsystem ist auch ein MP-geeigneter Scheduler nötig. Der zu Beginn dieser Arbeit vorhandene Scheduler war nicht für die Verwaltung mehrerer CPUs ausgelegt. Eine einfache Erweiterung um MP-Fähigkeit schien jedoch nicht zweckmäßig, da keine ausreichenden Möglichkeiten der Konfiguration gegeben waren.

Deshalb wurde ein neues Schedulingkonzept für JX entwickelt, wobei auf folgende Punkte besonderer Wert gelegt wurde:

- Unterstützung für Multiprozessorsysteme.
- Flexibilität bei der Realisierung einer Schedulingstrategie.
- Individuelle Verwaltung der Threads einer Domain.

Um einen flexiblen Scheduler zu realisieren, wurden einige zusätzliche Anforderungen gestellt:

- Es soll die Möglichkeit bestehen, die Schedulingstrategie leicht an eine gegebene Aufgabe anzupassen. Der Scheduler soll deshalb in Java implementiert sein.
- Der Scheduler soll zur Laufzeit beeinflussbar sein. Unter Umständen soll sogar das Auswechseln der Schedulingstrategie (zur Laufzeit) möglich sein.
- Die Wahl der Schedulingstrategie soll durch das Konzept nicht eingeschränkt werden. Das heißt, dass die Informationen, die das System dem Scheduler mitteilt, den Benutzer in die Lage versetzen soll, jede beliebige Strategie zu realisieren.
- Die Zeitscheibenverwaltung soll für jede Domain individuell einstellbar sein.

5.4 Konzept

5.4.1 Zweistufiges Grundkonzept

Ein wichtiges Konzept in JX ist die Unabhängigkeit der Domains. Um diesen Aspekt zu unterstützen, bietet sich ein zweistufiges Schedulingkonzept an, bei dem jede Domain ihre Threads selbst verwaltet – ähnlich wie bei Applikationsschedulern unter UNIX: Dort verwaltet der Kern die CPUs und stellt den Anwendungen Kernel-Threads zur Verfügung. Aufbauend auf diese Aktivitätsträger können Anwendungen ihre eigenen – so genannte User-Level-Threads – erzeugen, welche die Kernel-Threads als virtuelle Prozessoren nutzen. Die Verwaltung der Anwendungs-Threads übernimmt ein eigener Applikationsscheduler.

Unter UNIX besteht allerdings das Problem, dass der Kern nichts von der Existenz der Applikationsscheduler weiß. Er kann sie somit auch nicht in der Schedulingstrategie berücksichtigen. Wird durch einen Systemaufruf ein Kernel-Thread blockiert, so steht er dem Applikationsscheduler nicht mehr zur Verfügung. Handelt es sich dabei um den einzigen Kernel-Thread der Anwendung, so ist sie dadurch blockiert, auch wenn lauffähige User-Level-Threads vorhanden sind. In dem für JX entwickelten Konzept kann eine solche Situation nicht auftreten, da die Rechenzeit primär nicht an Aktivitätsträger, sondern an Applikationen (Domains) verteilt wird. Anwendungsspezifische Scheduler sind dem Kern somit bekannt.

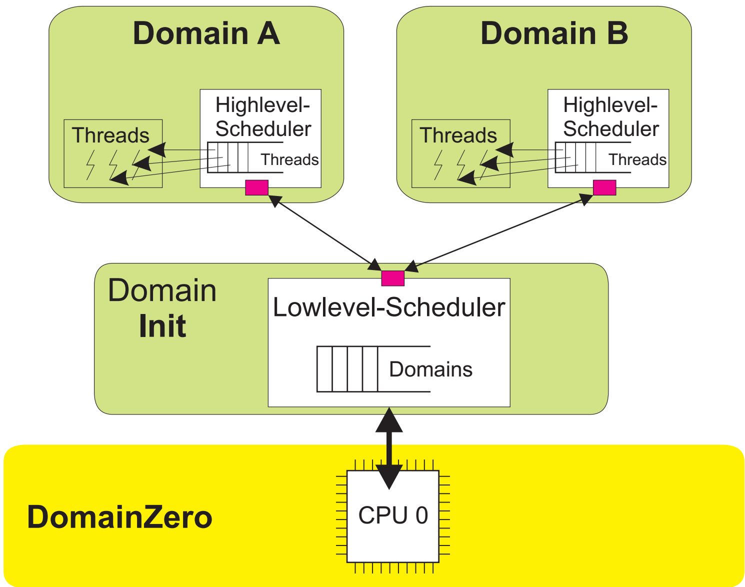


Abbildung 5.2: Zweistufiges Grundkonzept

Abbildung 5.2 zeigt den groben Aufbau des Konzepts. In der ersten Ebene erfolgt die Zuteilung der Rechenzeit an die Domains. Der Scheduler, welcher diese Aufgabe übernimmt, wird im Folgenden als *Lowlevel-Scheduler* bezeichnet. Jede Domain kann wiederum ihren eigenen, so genannten *Highlevel-Scheduler* erstellen, um die ihr zugeteilte Rechenzeit auf ihre Aktivitätsträger aufzuteilen.

Sowohl High- als auch Lowlevel-Scheduler sind in Java implementiert. Die Funktionalität eines Schedulerobjekts – und damit auch die Strategie der Threadverwaltung – kann somit frei definiert werden. Für die Anbindung an das System müssen lediglich bestimmte Schnittstellen implementiert werden. Auf die Interaktion mit Schedulerobjekten wird in Abschnitt 5.5 noch genauer eingegangen.

Durch den Lowlevel-Scheduler ist man nun in der Lage die Rechenzeit individuell auf die

Domains aufzuteilen. Meist wird wohl eine gleichberechtigte Zuteilung des Prozessors ausreichen, jedoch könnte man auch eine Art Priorität einführen, an Hand derer die Aufteilung der Rechenzeit erfolgt.

Die Highlevel-Scheduler ermöglichen jeder Domain, ihre Rechenzeit selbst aufzuteilen und eine Schedulingstrategie zu wählen, die ihren Anforderungen optimal entspricht. Da Highlevel-Scheduler normale Java-Objekte sind, kann eine Anwendung mit dem Objekt Informationen austauschen, um zur Laufzeit Einfluss auf die Verwaltung der Threads nehmen.

Um die Rechenzeit, die einer Domain zur Verfügung steht, auch individuell einsetzen zu können, wird das Konzept in Abschnitt 5.6 noch um eine individuelle Timer-Steuerung erweitert.

5.4.2 Unterstützung für Multiprozessorsysteme

Um mehrere Prozessoren zu unterstützen, ist im Konzept vorgesehen, dass für jeden Prozessor ein eigener Lowlevel-Scheduler eingesetzt wird. Die Verwaltung der Prozessoren kann somit unabhängig voneinander durchgeführt werden.

Auf der Ebene der Domains wird das Prinzip fortgesetzt: jedem Prozessor wird ein eigenes Schedulerobjekt zugeordnet. Eine Domain muss deshalb für jede CPU, die sie verwenden darf bzw. will, einen Highlevel-Scheduler beim Kern registrieren. Je nachdem, für welchen Prozessor die Domain Rechenzeit erhält, wird auch der entsprechende Scheduler angesprochen.

Abbildung 5.3 zeigt die Situation für ein System mit zwei Prozessoren. Domain B verwendet in diesem Beispiel beide CPUs.

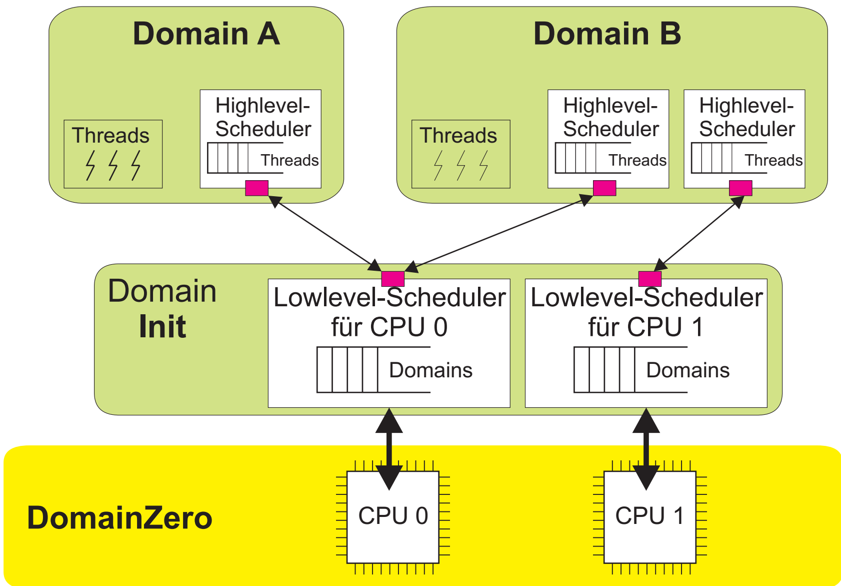


Abbildung 5.3: Zweistufiges Konzept für Multiprozessorsysteme

Getrennte Highlevel-Scheduler für alle verwendeten Prozessoren haben den Vorteil, dass die Verwaltungsstrukturen leicht konsistent zu halten sind. Implementiert jedes Schedulerobjekt eine eigene Warteschlange, dann müssen die Zugriffe nicht koordiniert werden, da ein Schedulerobjekt immer nur von einem Prozessor angesprochen wird. Es ist daher möglich, dieselbe Implementierung eines Highlevel-Schedulers einzusetzen wie im Monoprozessorrechner. Außerdem könnten für verschiedene Prozessoren auch verschiedene Schedulingstrategien realisiert werden.

Allerdings entstehen auch Nachteile, wenn die Prozessoren unabhängig voneinander verwaltet werden. Im Bereich der Lowlevel-Scheduler führt eine individuelle Verwaltung dazu, dass eine Domain die Prozessoren nicht unbedingt gleichzeitig nutzen darf. Im Allgemeinen arbeiten Anwendungen, die mehrere Prozessoren unterstützen, mit kooperierenden Aktivitätsträgern. Diese können unter Umständen nur optimal eingesetzt werden, wenn die Prozessoren gleichzeitig zur Verfügung stehen.

Es ist deshalb sinnvoll, wenn die Lowlevel-Scheduler miteinander kommunizieren, um ein so genanntes Co- oder Gang-Scheduling [Ous82] zu realisieren, welches die gleichzeitige Zuteilung der Prozessoren zum Ziel hat. Da die Lowlevel-Schedulerobjekte in Java implementiert sind und im Allgemeinen in der selben Domain liegen,[2] können zur Kommunikation einfache Methodenaufrufe genutzt werden.

Im Bereich der Highlevel-Scheduler kann eine separate Verwaltung der Prozessoren durch getrennte Warteschlangen zu einer unbalancierten Auslastung führen. Da Highlevel-Scheduler jedoch ebenfalls normale Java-Objekte sind, kann auch hier ein Informationsaustausch mittels Methodenaufrufen realisiert werden. Durch ein geeignetes Konzept können die Schedulerobjekte so Threads austauschen, um dadurch einen Lastausgleich zu erreichen.

5.4.3 Gemeinsames Schedulingobjekt für mehrere CPUs

Die Flexibilität des Schedulingkonzepts zeigt sich, wenn man die Kopplung zwischen Kern (Domain Zero) und Scheduler genauer betrachtet:

Um ein Highlevel-Schedulerobjekt einer Domain zuzuordnen, muss es beim Kern registriert werden. Dort wird dann eine Referenz auf das Objekt in den Domainkontrollblock eingetragen. Mittels der Objektreferenz kann die Domain Zero die gewünschten Methoden eines Schedulers aufrufen. Wird z. B. ein Thread blockiert, so wird zuerst festgestellt, welcher Domain er angehört. Daraufhin kann das entsprechende Schedulerobjekt angesprochen und über das Ereignis informiert werden.

Im Multiprozessorbetrieb wird im Domainkontrollblock für jeden Prozessor eine Referenz auf ein Schedulerobjekt verwaltet. Registriert man nun in einer Domain für jeden Prozessor dasselbe Schedulerobjekt, so wird für alle CPUs ein Verweis auf denselben Highlevel-Scheduler eingetragen. Das hat zur Folge, dass für alle Prozessoren das gleiche Schedulerobjekt angesprochen wird.

[2] In Abbildung 5.3 sind die Lowlevel-Schedulerobjekte in der Domain Init dargestellt. Prinzipiell könnten die Lowlevel-Scheduler jedoch in jeder beliebigen Domain liegen.

Eine Lastverteilung ist daher leicht durch eine gemeinsame Warteschlange zu realisieren. Bei der Implementierung des Schedulers muss dann allerdings auf eine geeignete Absicherung der Datenstrukturen geachtet werden, da der Kern keinen gegenseitigen Ausschluss gewährleistet.

Auch im Bereich der Lowlevel-Scheduler kann ein Schedulerobjekt für mehrere CPUs registriert werden. Allerdings wurde in Abschnitt 3.4 erwähnt, dass man Domains auf bestimmte Prozessoren beschränken kann. Daraus folgt, dass nicht jede Domain für alle CPUs einen Highlevel-Scheduler registriert hat. Die Realisierung eines gemeinsamen Lowlevel-Schedulers für alle Prozessoren muss auf diese CPU-Beschränkung achten und kann daher relativ komplex werden.

5.4.4 Externes Scheduling

Für bestimme Ressourcen, wie zum Beispiel das Speichermanagement, bietet sich eine externe Verwaltung an. Bei der Speicherverwaltung soll ein Garbage-Collector den Speicher aufräumen. Das wird jedoch erschwert, wenn er selbst auf dem ihm unterstellten Speicher arbeitet. Es ist somit sinnvoller den Garbage-Collector und die Speicherverwaltung nicht in der Domain anzulegen, welcher der Speicher zugeordnet ist.

Vergleichbare Situationen können auch bei der Verwaltung der Threads auftreten. Das Schedulingkonzept sieht es beispielsweise ausdrücklich vor, dass ein Schedulerobjekt zur Laufzeit durch ein anderes ausgetauscht werden kann, um die Schedulingstrategie zu ändern.

Durch die Verwaltung der Scheduler – wie oben beschrieben – ist das Austauschen des Schedulerobjekts selbst relativ einfach. Da nicht jeder Thread einen Verweis auf seinen Scheduler hat, muss lediglich der Eintrag im Domainkontrollblock atomar verändert werden. Aus der Sicht des Systems ist somit der neue Scheduler für die Domain zuständig. Er wird deshalb von nun an über die Zustandswechsel der Threads der Domain informiert.

Bevor der Scheduler jedoch seine Arbeit aufnehmen kann, müssen noch die Informationen des alten Schedulerobjekts in den neuen Scheduler übertragen werden. Wird nun der Thread, der diese Aufgabe übernimmt, verdrängt, so kann er unter Umständen nicht mehr aktiviert werden, da der neue Scheduler noch nichts von seiner Existenz weiß.

Abhilfe kann man schaffen, indem zum Beispiel das gesamte System angehalten wird, bis der Austausch des Schedulers – incl. der Verwaltungsdaten – abgeschlossen ist. Der Thread, der die Änderung durchführt, kann somit nicht verdrängt werden. Nachteil dieses Ansatzes ist jedoch, dass sich eine Domain – unter dem Vorwand das Schedulerobjekt auszutauschen – Rechenzeit aneignen kann, die ihr nicht zusteht.

Eine sinnvolle Alternative ist, die Verwaltung der Aktivitätsträger von einer anderen Domain aus durchzuführen. Der Austausch des Schedulers nimmt dann ein Thread der verwaltenden Domain vor. Die betroffene Domain kann dadurch "eingefroren" werden, bis die Verwaltungsdaten übertragen wurden.

Abbildung 5.4 zeigt eine Domain, welche die Threads von zwei anderen Domains verwaltet. Dazu enthält sie je ein zusätzliches Schedulerobjekt. Darüberhinaus kann sie jedoch auch noch

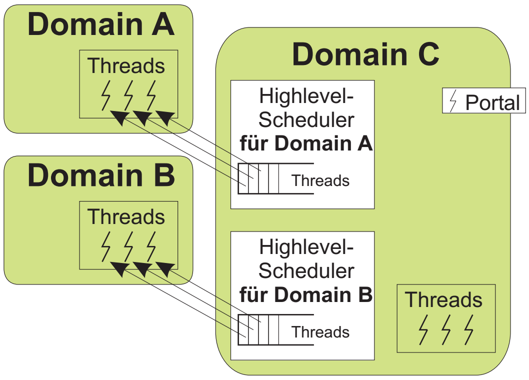


Abbildung 5.4: Threadverwaltung in externer Domain

beliebige andere Aufgaben erfüllen, wenn sie – vom Lowlevel-Scheduler – Rechenzeit zugeteilt bekommt.

Das Verfahren beim Scheduling unterscheidet sich in dieser Konfiguration nicht vom normalen Vorgehen, wie es später in Abschnitt 5.5 erläutert wird: Ein Lowlevel-Scheduler, der beispielsweise der Domain A Rechenzeit geben möchte, wendet sich an den Highlevel-Scheduler, der im Domainkontrollblock (von A) eingetragen ist. Dabei ist es egal, in welcher Domain dieses Objekt tatsächlich liegt. Der Highlevel-Scheduler kann anschließend einen lauffähigen Thread (von Domain A) auswählen und ihn aktivieren.

Um der betroffenen Domain eine Konfiguration ihres Schedulers zu ermöglichen, könnte die Domain C ein Portal zur Verfügung stellen. Domain B könnte im obigen Beispiel durch einen entsprechenden Portalaufruf veranlassen, dass ihr Schedulerobjekt ausgewechselt wird. Der Austausch selbst wird daraufhin jedoch von der Domain C vorgenommen.

Durch die Auslagerung des Schedulers in eine andere Domain entstehen allerdings zwei Nachteile:

1. Die beiden beteiligten Domains sind nicht mehr unabhängig voneinander.
2. Eine Kommunikation zwischen der Anwendung und ihrem Scheduler ist teurer, da domainübergreifende Methodenaufrufe eingesetzt werden müssen.

Zumindest der erste Punkt kann im Allgemeinen entkräftet werden, da die beteiligten Domains üblicherweise von einem Entwickler erstellt werden, sodass eine genaue Kenntnis über den Code der jeweils anderen Domain vorhanden ist.

5.5 Kommunikation mit Schedulern

Nach der Beschreibung des hierarchischen Konzeptes sollen nun die Abläufe zwischen den Schedulerobjekten bzw. zwischen Kern und Schedulerobjekten betrachtet werden.

5.5.1 Einführung

Bei der zweistufigen Verwaltung der Rechenzeit müssen häufig Informationen zwischen den beiden Ebenen ausgetauscht werden. Der Lowlevel-Scheduler einer CPU muss daher in vielen Fällen mit den Highlevel-Schedulern in den Domains kommunizieren, zum Beispiel wenn ein Thread deblockiert oder verdrängt wurde.

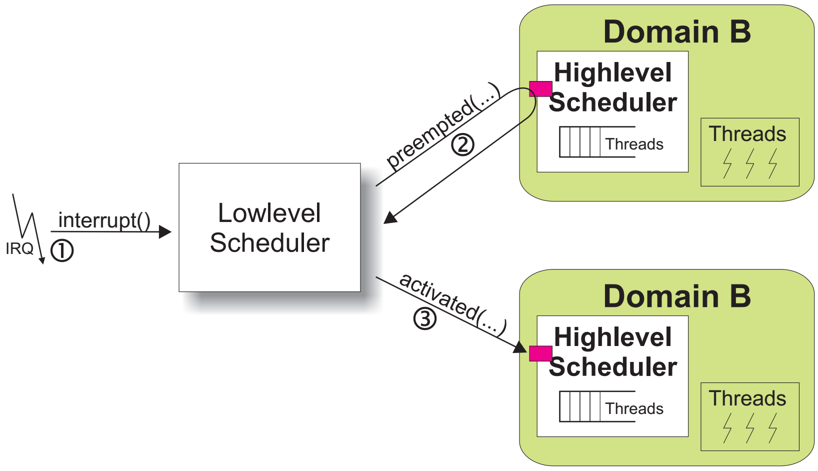


Abbildung 5.5: Kommunikation zwischen den Schedulern am Beispiel eines Domainwechsels

In Abbildung 5.5 ist der Informationsaustausch bei einem zeitscheibenbedingten Domainwechsel dargestellt: Zuerst wird der Lowlevel-Scheduler von der Domain Zero informiert, dass ein Timer-Interrupt eingetroffen ist (1). Daraufhin führt er einige Überprüfungen durch und kommt zu dem Ergebnis, dass die Zeitscheibe der aktuellen Domain abgelaufen ist und deshalb ein Domainwechsel stattfinden muss. Die momentan aktive Domain wird daher zuerst darüber informiert, dass sie verdrängt wurde (2). Der entsprechende Highlevel-Scheduler hat dadurch die Gelegenheit beispielsweise den Zeitpunkt der Verdrängung zu notieren. Als nächstes wird der Lowlevel-Scheduler eine neue Domain auswählen und diese aktivieren (3).

5.5.2 Benachrichtigungsmöglichkeiten

Wie in Abbildung 5.5 bereits angedeutet, stellt ein Highlevel-Scheduler Methoden zur Verfügung, die vom Lowlevel-Scheduler oder dem Kern aufgerufen werden, um Ereignisse – beispielsweise die Verdrängung oder Blockierung eines Threads – zu melden. Bevor der Ablauf genauer untersucht wird, soll zunächst eine Übersicht über alle Methoden gegeben werden, die ein Highlevel-Scheduler zu diesem Zweck zur Verfügung stellen kann (siehe auch Anhang A.2).

In Tabelle 5.1 sind diese Methoden aufgezählt. Dabei werden sie in Gruppen eingeteilt:

- Zum einen in Informationsmethoden, die den Highlevel-Scheduler über Ereignisse informieren, die einen seiner Threads betreffen. Das Ereignis ist implizit durch den Methodenaufruf definiert, der Thread wird als Parameter erwartet. In der Tabelle ist diese Gruppe weiter unterteilt, in:

- ◦ Methoden, die einen lauffähigen Thread als Parameter erhalten und
- ◦ Methoden, deren Parameter ein nicht (mehr) lauffähiger Thread ist.

- Die zweite Gruppe bezeichnet Methoden, die zur Aktivierung der Domain eingesetzt werden. Eine genaue Erklärung dieser Gruppen folgt im nächsten Abschnitt.

Informationsmethoden:

Methode des HLS	wird aufgerufen wenn ...
preempted	... ein Thread verdrängt wurde, weil die Zeitscheibe der Domain abgelaufen ist.
interrupted	... ein Thread durch die interne Zeitscheibe einer Domain unterbrochen wurde.
created*	... ein Thread erzeugt wurde.
unblocked	... ein zuvor blockierter Thread wieder lauffähig geworden ist.
yielded	... ein Thread seine Rechenzeit freiwillig abgeben möchte.
destroyed	... sich ein Thread beendet hat.
blocked	... sich ein Thread blockiert hat.
blockedInPortalCall	... ein Thread in einem Portalaufruf blockiert wurde.
portalCalled	... ein Thread in einem Portalaufruf blockiert wurde und der Scheduler die Möglichkeit hat, die verbleibende Rechenzeit der Domain des Portals zu überlassen.

Aktivierungsmethoden:

Methode des HLS	wird aufgerufen wenn ...
activated*	... der Scheduler einen Thread aktivieren darf.
switchTo	... ein bestimmter Thread einer Domain aktiviert werden soll.

die mit * gekennzeichneten Methoden sind obligatorisch

Tabelle 5.1: Mögliche Benachrichtigungen eines Highlevel-Schedulers

Im Folgenden soll die Bedeutung jeder Methode kurz erklärt werden:

preempted: zeigt an, dass die Rechenzeit der Domain verbraucht ist. Der momentan aktive Thread wurde deshalb unterbrochen. Der aktiven Domain wird dieser Thread als Parameter mitgeteilt.

interrupted: Diese Methode wird aufgerufen, wenn die interne Zeitscheibe einer Domain angelaufen ist.[3] Sie entspricht somit der Methode preempted mit dem Unterschied, dass anschließend die gleiche Domain wieder aktiviert wird.

[3]In Abschnitt 5.6 wird noch gezeigt, wie eine Domain eine interne Zeitscheibe definieren kann.

created: wird aufgerufen, wenn ein neuer Thread erzeugt wurde. Der Highlevel-Scheduler kann ihn in seine Warteschlange der lauffähigen Threads einhängen.

unblocked: hat einen lauffähigen Thread als Parameter, der zuvor blockiert wurde. Das heißt, der Scheduler wurde zuvor durch die Methode blocked, blockedInPortalCall oder portalCalled darüber informiert, dass dieser Thread blockiert ist. Falls ein Scheduler auch die blockierten Threads in einer Warteschlange verwaltet, so kann er ihn nun als lauffähig markieren.

yielded: zeigt an, dass ein Thread seine Rechenzeit freiwillig abgeben möchte, z. B. durch den Aufruf von java.lang.Thread::yield(). Der Thread bleibt dadurch allerdings lauffähig.
Im Anschluss an diese Methode erfolgt sofort ein Aufruf der Methode activated, da die Domain den Prozessor weiterhin nutzen darf. Der Scheduler sollte dann allerdings einen anderen Thread aktivieren. Existiert kein anderer Thread in dieser Domain, so ist es meist sinnvoll, dass die Domain ihre Rechenzeit abgibt, anstatt den Thread erneut zu aktivieren.

destroyed: wird aufgerufen, wenn ein Thread beendet wurde. Er darf danach nicht mehr aktiviert werden.

blocked: zeigt an, dass ein Thread blockiert wurde. Der Thread ist daraufhin nicht mehr lauffähig.

blockedInPortalCall: informiert den Scheduler, dass ein Thread einen Portalaufruf gestartet hat, das Portal aber nicht "frei" war. (Portale werden in Kapitel 6 beschrieben.) Das heißt, es bestand nicht die Möglichkeit Rechenzeit abzugeben. Der Thread wurde deshalb blockiert, bis das Portal frei wird und der Aufruf bearbeitet wurde.

portalCalled: äquivalent zu blockedInPortalCall, mit dem Unterschied, dass das Portal frei ist. Diese Methode erwartet als Rückgabe, ob sofort eine Umschaltung auf den Portalthread durchgeführt werden soll oder nicht.

activated: in dieser Methode ist üblicherweise die Schedulingstrategie implementiert. Die Methode dient als Startpunkt für den Highlevel-Scheduler, wenn eine Domain aktiviert wird.

switchTo: auch hierdurch wird die Kontrolle an die Domain übergeben. Allerdings wird ein Thread als Parameter mitgegeben, der nach Möglichkeit aktiviert werden soll. Der Scheduler behält jedoch die Kontrolle darüber, welcher Thread tatsächlich aktiviert wird.

5.5.3 Interaktion mit Highlevel-Schedulern

Informationsmethoden

Wie schon kurz erläutert, sind die meisten Methoden in Tabelle 5.1 Informationsmethoden, um einem Highlevel-Scheduler den Zustandswechsel eines Threads anzuzeigen.

Diese Methoden werden bei deaktiviertem Timer-Interrupt ausgeführt, um Probleme durch eine Unterbrechung zu vermeiden. Denn würde beispielsweise die Methode preempted von einem Timer-Interrupt verdrängt werden, so müsste die entsprechende Domain darüber informiert werden. Diese Situation könnte in eine Schleife führen, die unter Umständen nicht aufzulösen ist. Bei den anderen Methoden liegt der Grund darin, dass der Scheduler die Möglichkeit haben soll, den Zeitpunkt des Ereignisses aufzeichnen zu können, ohne vorher verdrängt zu werden.

Da die Informationsmethoden nicht unterbrochen werden, könnte eine Domain jedoch sehr viel Rechenzeit verbrauchen, wenn dort übermäßig viel Code ausgeführt wird. Ein Bytecode-Verifier [Alt01] überprüft deshalb die Laufzeit dieser Methoden – entweder statisch vor der Übersetzung des Bytecodes in Maschinencode oder zur Laufzeit – und sorgt für die Einhaltung einer oberen Schranke.

Die Implementierung dieser Informationsmethoden muss deshalb auf wenige Aktionen beschränkt sein. Im Allgemeinen sollte lediglich der Status eines Aktivitätsträgers verändert, ein Zeitstempel notiert oder der Thread in eine Warteschlange eingehängt werden.

Aktivierungsmethoden

Neben den Informationsmethoden ist noch eine zweite Gruppe von Methoden vorgesehen, die der Highlevel-Scheduler implementieren kann bzw. muss. Diese besteht aus den Methoden activated und switchTo. Sie werden ausgeführt, wenn eine Domain einen Aktivitätsträger aktivieren darf.

Um einer Domain Rechenzeit zuzuweisen, wird ein ähnlicher Ansatz gewählt wie in [FS96] oder [KR94] vorgeschlagen. Will der Lowlevel-Scheduler eine Domain aktivieren, so gibt er die Kontrolle über den Prozessor an den entsprechenden Highlevel-Scheduler ab. Dazu verfügt jeder Highlevel-Scheduler über einen (Scheduling-)Thread. Dieser Aktivitätsträger führt die Methode activated aus, in der die Schedulingstrategie hinterlegt sein soll. Hat diese einen lauffähigen Thread der Domain ausgewählt, so wird er aktiviert. Somit wird die Rechenzeit, die dem Highlevel-Scheduler übertragen wurde an einen Thread der Domain weitergegeben.

Im Gegensatz zu dem in [FS96] vorgeschlagenen Ansatz, bei dem der Schedulingthread gesichert wird, wenn er die CPU-Zeit an einen Thread abgibt, wird er hier durch den neu aktivierten Aktivitätsträger "ersetzt", d. h. der Zustand des Schedulingthreads wird nicht gesichert. Wird die Domain in der nächsten Zeitscheibe wieder aktiviert, so wird der Schedulingthread erneut mit der Methode activated gestartet. Vorteil dabei ist, dass die Zeit für das Sichern des Schedulingthreads eingespart werden kann. Allerdings müssen Vorkehrungen getroffen werden, falls der Thread verdrängt wird bevor die Schedulingstrategie beendet wurde; Abschnitt 5.5.7 geht auf diese Situation ein.

Aus Sicht der Systems bzw. des Lowlevel-Schedulers wurde durch das Aktivieren des Highlevel-Schedulers die Domain bereits aktiviert. Die Zeit die für die Schedulingstrategie benötigt wird, kann der Lowlevel-Scheduler deshalb schon der "neuen" Domain zuordnen. Es kann dadurch nicht vorkommen, dass eine Domain durch einen fehlerhaften oder böswilligen Scheduler Rechenzeit verbraucht, die ihr nicht zusteht.

Der Schedulingthread wird auch im Rahmen des Portalmechanismus verwendet, wenn ein bestimmter Thread der Domain aktiviert werden soll. Dazu ist die Methode switchTo vorgesehen. Genauere Informationen sind in Abschnitt 6.5.2 zu finden.

Zusammenfassung

Zusammenfassend ist festzuhalten, dass man die Interaktion mit einem Highlevel-Scheduler in zwei Arten einteilen kann:

- Zum einen Mitteilungen, welche den Highlevel-Scheduler über Ereignisse informieren, die einen von ihm verwalteten Thread betreffen. Die entsprechenden Methoden – d. h. die Reaktion auf diese Ereignisse – werden unter Interruptausschluss aufgeführt und müssen deshalb zeitlich begrenzt sein.

- Zum anderen die Zuteilung von Rechenzeit zu einer Domain, d. h. die Aufforderung einen Thread zu aktivieren. Die entsprechenden Methoden müssen nicht zeitlich begrenzt sein. Da der Aufwand der Schedulingstrategie jedoch beliebig groß sein kann, werden diese Methoden bereits in der Rechenzeit der Domain ausgeführt.

5.5.4 Funktionsaufrufe über Domaingrenzen

Problemstellung

Im Beispiel in Abbildung 5.5 muss der Lowlevel-Scheduler mit zwei unterschiedlichen Highlevel-Schedulern kommunizieren. Da die beteiligten Objekte im Allgemeinen jedoch nicht in derselben Domain liegen, müssen Methodenaufrufe über Domaingrenzen hinweg durchgeführt werden.

Für diesen Zweck werden normalerweise Portale eingesetzt. Der Portalmechanismus setzt jedoch ein funktionierendes Schedulersystem voraus,[4] sodass er hierfür nicht genutzt werden kann.

Die Methoden eines Objekts in einer anderen Domain können aber auch nicht direkt aufgerufen werden, da Objekte die zur Ausführung benötigt werden, nicht erreichbar wären. Ein Threadwechsel auf einen Aktivitätsträger der Zieldomain ist daher unumgänglich.

[4] Während eines Portalaufrufes müssen Threads blockiert oder aktiviert werden. Daher ist eine Interaktion mit den entsprechenden Schedulern notwendig. In Kapitel 6 sind genauere Informationen über einen Portalaufruf zu finden.

Schedulerportale

Die Umschaltung auf einen anderen Aktivitätsträger ist normalerweise relativ teuer. Zuerst müssen alle Registerinhalte des aktuellen Threads gesichert werden, anschließend müssen die Inhalte aus dem Threadkontrollblock des neuen Aktivitätsträgers in die Register geladen werden.

In dem speziellen Fall, dass lediglich die Ausführung einer Funktion in einen anderen Thread "ausgelagert" werden muss, können jedoch einige besondere Eigenschaften ausgenutzt werden, auf die im Folgenden eingegangen wird.

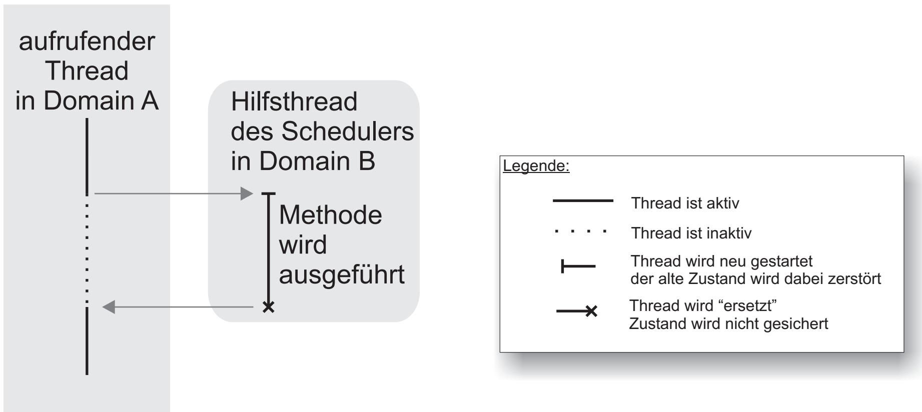


Abbildung 5.6: Funktionsweise eines Schedulerportals

In der vorliegenden Situation soll eine Methode eines Highlevel-Schedulers in einem anderen Thread ausgeführt werden. Abbildung 5.6 gibt eine schematische Darstellung der Vorgehensweise. Als Voraussetzung muss ein inaktiver, unbenutzter Hilfsthread in der Zieldomain vorhanden sein. Dieser wird zuerst für die Ausführung der Funktion vorbereitet: in seinem Threadkontrollblock wird zum einen die Startadresse auf den Code der Methode gesetzt. Zum anderen wird der Stack vorbereitet. Dabei müssen folgende Schritte durchgeführt werden:

- Als erstes Element des Stacks wird die Rücksprungadresse abgelegt. An dieser Stelle wird die Ausführung fortgesetzt, nachdem die Methode beendet ist. Es handelt sich deshalb um die Adresse einer Funktion, die auf den aufrufenden Thread zurückschaltet.
- Als nächstes wird die Adresse des Objektdeskriptors abgelegt, an dem die Methode aufgerufen werden soll.
- Benötigt die Methode Parameter, so werden diese anschließend auf den Stack des Threads gelegt.

Nachdem der Thread vorbereitet ist, kann die Aktivierung bzw. die Umschaltung erfolgen. Dabei kann man sich zunutze machen, dass der aufrufende Thread an einem Funktionsaufruf unterbrochen wird. Es müssen deshalb nicht alle Register gesichert werden, wie es bei einer

Unterbrechung an beliebiger Stelle (z. B. durch einen Interrupt) der Fall wäre. Stattdessen genügt es, die Register zu sichern, deren Inhalt eine Funktion nicht verändern darf.[5] In diesen könnten Daten abgelegt sein, die über einen Funktionsaufruf hinweg erhalten bleiben müssen. Nur sie sind somit für die weitere Ausführung des (alten) Threads notwendig.

Beim Aktivieren des neuen Threads müssen ebenfalls nicht alle Register aus dem Threadkontrollblock in den Prozessor geladen werden, da die Ausführung am Anfang einer Funktion beginnt. Der Inhalt der meisten Register darf beim Start einer Funktion beliebig sein, da deren Inhalt nicht verwendet wird. Für die Aktivierung des neuen Threads müssen daher nur der Instruktionszeiger (EIP) und der Stackzeiger (ESP) in den Prozessor geladen werden.

Im Gegensatz zu einer kompletten Threadumschaltung an beliebiger Stelle, müssen in diesem speziellen Fall statt aller 13 Register des Prozessors[6] nur 7 Register gesichert werden und nur zwei statt 13 geladen werden.

Hat der Hilfsthread die Methode beendet, so wird an der zuvor auf dem Stack festgelegten Rücksprungadresse fortgefahren. Der dort abgelegte Code sorgt für die Wiederherstellung des aufrufenden Threads. Dabei müssen nicht alle, sondern nur die zuvor gesicherten 7 Registerinhalte wieder hergestellt werden.

Vom aktuellen Thread – also dem, der die Methode ausgeführt hat – müssen keine Register gesichert werden, da sie keine relevanten Informationen enthalten. Lediglich der Rückgabewert muss zum Aufrufer übertragen werden. Das heißt, dass der Inhalt des Registers EAX während der Rückschaltung zum aufrufenden Threads erhalten bleiben muss.

Geschwindigkeitsvergleich

	Methodenaufruf durch		
	Portal	Schedulerportal	direkt
Zeit pro Aufruf	1037 ns	428 ns	88 ns

gemessen auf einem 500MHz Pentium III System

Tabelle 5.2: Geschwindigkeitsvergleich zwischen Portal und Schedulerportal

Tabelle 5.2 zeigt die Unterschiede zwischen einem Methodenaufruf durch ein Portal und mit Hilfe des eben vorgestellten Mechanismus – im Folgenden Schedulerportal genannt. Der Vergleich mit einem Threadwechsel an beliebiger Stelle ist nicht aufgeführt, da er keine wirkliche Alternative darstellt. Denn sowohl die Umschaltung auf den Hilfsthread als auch die Rückschaltung, finden immer an definierten Stellen statt. Der Portalaufrufmechanismus nutzt diesen Vorteil ebenfalls aus und ist dadurch der bessere Vergleichspartner.

[5] Um welche Register es sich handelt, bestimmen die Aufrufkonventionen des verwendeten Compilers. Beim GNU C-Compiler für i386 Prozessoren – der für die Übersetzung des Kerns eingesetzt wird – darf ein Funktionsaufruf folgende Register nicht verändern: den Instruktionszeiger (EIP), den Stackzeiger (ESP), den Basepointer (EBP), das Register EBX, die Indexregister ESI und EDI und die Prozessorflags (EFLAGS).

[6] Es handelt sich dabei im Einzelnen um GS, FS, ES, EDI, ESI, EBP, EAX, EBX, ECX, EDX, EIP, EFLAGS und ESP.

Man kann feststellen, dass ein Aufruf durch den Schedulerportalmechanismus mehr als doppelt so schnell ist, als ein Portal. Das kommt nicht nur dadurch zustande, dass weniger Register gesichert werden, sondern auch durch einen geringeren Verwaltungsaufwand.

Verglichen mit einem direkten Aufruf einer virtuellen Methode an einem beliebigen lokalen Objekt, ist der Schedulerportalmechanismus jedoch fast fünfmal langsamer.

5.5.5 Geschwindigkeitsvergleich verschiedener Konzepte

Um den Einfluss der Kommunikation auf die Schedulinggeschwindigkeit zu erfassen, wurde die Zeit einer Threadumschaltung bei verschiedenen Schedulingkonzepten gemessen. Tabelle 5.3 zeigt die Ergebnisse. Es wurden drei unterschiedliche Situationen betrachtet:

- Im ersten Versuch ("destroy") wurde untersucht wie lange es dauert, bis ein Thread beendet und zum nächsten umgeschaltet wird. Das heißt, ein Thread notiert als letzte Operation einen Zeitstempel. Dann beendet er sich woraufhin ein anderer Aktivitätsträger aktiviert wird. Die erste Aktion des neuen Threads besteht darin, ebenfalls einen Zeitstempel zu notieren. Die Differenz der Zeiten ist das Ergebnis der Messung.

- Im zweiten Versuch ("yield") wird die Situation dahingehend verändert, dass der erste Thread nicht beendet wird, sondern seine Rechenzeit freiwillig abgibt.[7] Da in diesem Fall der Kontext des Aktivitätsträgers gesichert werden muss und der Thread wieder an den Scheduler zurückgegeben wird, ist die Zeitdifferenz grundsätzlich etwas größer.

- Bei der letzten Messung ("interrupt") wurde der Versuchsaufbau nochmals verändert. Der erste Thread wird nun durch einen Timer-Interrupt verdrängt. Dazu wurde der Zeitgeber auf ein definiertes Intervall eingestellt und gleichzeitig ein Zeitstempel notiert. Anschließend wurde gewartet, bis der aktive Thread verdrängt wird und auf ein anderer aktiviert wurde.

	destroy	yield	interrupt
globale Warteschlange (in C)	0,66 μs	0,80 μs	1,25 μs
globaler Scheduler (in Java)	1,40 μs	2,08 μs	2,06 μs
zweistufiger Scheduler (in Java)	2,36 μs	2,50 μs	3,11 μs

Tabelle 5.3: Geschwindigkeitsvergleich zwischen verschiedenen Schedulingkonzepten

Diese drei Messungen wurden mit drei unterschiedlichen Schedulingkonzepten durchgeführt. Die erste Zeile zeigt die Ergebnisse bei der Verwendung des ursprünglichen Schedulings von JX. Wie in Abschnitt 5.2 beschrieben, handelt es sich dabei um einen globalen Scheduler, der im Kern implementiert ist.

[7] Dazu kann die statische Methode java.lang.Thread::yield() aufgerufen werden.

Die Messungen in der zweiten Zeile wurden wiederum an einem globalen Scheduler durchgeführt. Allerdings wurde ein Schedulingobjekt eingesetzt welches in Java implementiert war.[8] Wie zu sehen ist, benötigt eine Threadumschaltung in allen drei Situationen deutlich länger. Dies ist vor allem darauf zurückzuführen, dass die Kommunikation zwischen Kern und Scheduler domainübergreifend ist und daher mit einer Umschaltung auf einen Hilfsthread verbunden ist.

In der letzten Zeile sind die Messergebnisse des zweistufigen Konzepts vorgestellt, welches im Rahmen dieser Arbeit entwickelt wurde. Im Gegensatz zu den beiden anderen Konzepten wird hier dem Scheduler mitgeteilt, dass ein Thread beendet wurde. Deshalb ist die Differenz zwischen dem ersten Versuch (destroy) und dem zweiten (yield) relativ gering (vgl. Tabelle 5.4). Beim dritten Test (interrupt) bemerkt man einen deutlich größeren Zeitverbrauch als bei den anderen Konzepten. Das rührt daher, dass der Lowlevel-Scheduler zuerst die Unterbrechung des aktuellen Threads mitteilt und anschließend erst die Kontrolle an den Schedulingthread abgibt (siehe auch Abbildung 5.5).

5.5.6 Konfiguration der Benachrichtigungen

Da nicht jede Schedulingstrategie alle möglichen Informationen benötigt und zudem domainübergreifende Methodenaufrufe relativ teuer sind, wurde das Benachrichtigungskonzept konfigurierbar gestaltet.

Besonderer Wert wurde dabei auf folgende Punkte gelegt:

- Individuelle Wahl der Benachrichtigungen.
- Steigerung der Geschwindigkeit durch Einschränkung der Benachrichtigungen.

Zur Implementierung eines Schedulers werden mehrere Java-Interfaces zur Verfügung gestellt (siehe Anhang A.2). Ein Basisinterface enthält die obligatorischen Methoden, die jeder Scheduler implementieren muss. Das sind, neben einigen Methoden zur Installation und Fehlerrückmeldung, hauptsächlich die Methoden `activated` und `created`. Alle anderen Methoden werden in einzelnen Interfaces definiert, die optional implementiert werden können.

Die Konfiguration wird implizit durchgeführt: Wird ein Interface von einem Scheduler implementiert, so wird er über das entsprechende Ereignis informiert; wird ein Interface nicht implementiert, entfällt auch die Benachrichtigung.

Ein Scheduler muss somit nur das Basisinterface implementieren. Dieses reicht aus, um die Funktionsfähigkeit (aus der Sicht des Systems) zu gewährleisten. Der Zustandswechsel eines Threads wird dem Schedulerobjekt dann zwar nicht gemeldet, um den Status eines Aktivitätsträgers (lauffähig, blockiert, ...) zu erfahren, kann er aber das Threadobjekt direkt abfragen (siehe Anhang A.3). Die meisten Schedulingstrategien werden jedoch nicht ohne das eine oder andere zusätzliche Interface auskommen.

[8] Dieses Konzept bildete eine Vorstudie bei der Entwicklung des zweistufigen Konzepts.

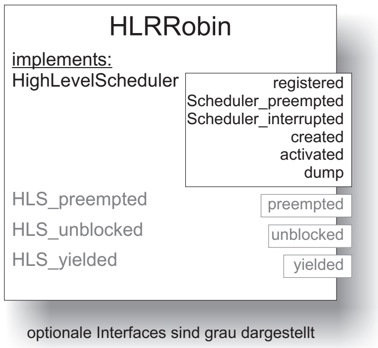


Abbildung 5.7: Realisierte Interfaces eines einfachen Round-Robin-Schedulers

In Abbildung 5.7 wird die Konfiguration eines Round-Robin-Schedulers skizziert. Das Interface HighlevelScheduler kennzeichnet das Objekt als Scheduler. Es enthält die Funktion activated und einige weitere zwingend vorgeschriebene Methoden.

Der Beispiel-Scheduler soll eine Warteschlange der inaktiven, lauffähigen Threads verwalten. Deshalb muss er über alle Ereignisse informiert werden, durch die ein Thread lauffähig wird oder lauffähig bleibt, aber inaktiv wird. Er muss daher informiert werden, wenn:

- ein neuer Thread erzeugt wird (Interface: HighLevelScheduler),
- ein Thread verdrängt wird (Interface: HLS_preempted),
- ein blockierter Thread lauffähig wird (Interface: HLS_unblocked) und wenn
- ein aktiver Thread freiwillig die Kontrolle abgibt (Interface: HLS_yielded).

Damit der Highlevel-Scheduler auf die gewünschten Ereignisse reagieren kann, müssen die entsprechenden Interfaces und somit die Informationsmethoden implementiert werden.

Um die Konfiguration übersichtlicher zu gestalten, könnte man die Benachrichtigungsmöglichkeiten in sinnvolle Pakete zusammenstellen. Auf die Möglichkeit einzelne Informationsmethoden individuell wählen zu können, soll jedoch nicht verzichtet werden, da zu viele unterschiedliche Implementierungen von Schedulern denkbar sind.

zweistufiges Konzept	Test: “destroy“
mit Benachrichtigung	2,36 μs
ohne Benachrichtigung	2,01 μs

Tabelle 5.4: Geschwindigkeitsgewinn durch “Abschalten” einer Benachrichtigung

Der Geschwindigkeitsgewinn, der durch “Abschalten” einiger Informationsquellen erreicht werden kann, wird durch Tabelle 5.4 belegt. Gemessen wurde die Zeit einer Threadumschaltung, wobei der ursprüngliche Thread beendet wurde (vgl. Abschnitt 5.5.5). Dabei kam zum einen der oben skizzierte Round-Robin-Scheduler zum Einsatz, zum anderen wurde zusätzlich die

Informationsmethode destroyed implementiert. Dadurch wird der Scheduler informiert, dass einer seiner Threads beendet wurde. Zu erkennen ist, dass der Verzicht auf eine Benachrichtigung in etwa die Zeit eines Schedulerportals einspart (vgl. Tabelle 5.2).

In Tabelle 6.2 (in Abschnitt 6.8) ist ein weiteres Beispiel dargestellt, wie durch den Verzicht auf Benachrichtigungen Zeit gespart werden kann.

5.5.7 Implementierung

In diesem Abschnitt werden einige Implementierungsdetails vorgestellt. Außerdem werden einige Ansätzte zur Optimierung vorgeschlagen. Folgende Punkte werden betrachtet:

- Hilfsthreads der Highlevel-Scheduler.
- Speicherverbrauch der Hilfsthreads.
- Aufruf von Schedulerportalen.

Hilfsthreads für Schedulerportale

Für die Realisierung des in Abschnitt 5.5.4 beschriebenen Aufrufmechanismus müssen freie, unbenutzte Threads zur Verfügung stehen. Bei der Registrierung eines Highlevel-Schedulers werden deshalb zwei Threads angelegt.

Einer der Threads wird verwendet, um die Informationsmethoden, die unter Interruptausschluss ablaufen, abzuwickeln. Da der Aktivitätsträger nicht unterbrochen werden kann und der Code, der von dem Thread ausgeführt wird, auch keine weiteren domainübergreifenden Methoden (z. B. Portale) aufrufen darf,[9] ist sichergestellt, dass der Thread immer unbenutzt ist, wenn eine entsprechende Methode aufgerufen werden soll.

Der zweite bei der Installation des Schedulers erzeugte Thread, wird bei der Aktivierung der Domain eingesetzt. Er wurde in Abschnitt 5.5.3 als Schedulingthread bezeichnet. Wie dort erwähnt, führt er die Methode activated aus, wenn die Kontrolle einem Highlevel-Scheduler übergeben werden soll. Da von dieser Methode keine Laufzeitbeschränkung gefordert wird, könnte es bei einer sehr aufwändigen – oder fehlerhaft implementierten – Schedulingstrategie vorkommen, dass die Zeitscheibe der Domain abgelaufen ist, bevor der Scheduler auf einen anderen Thread umgeschaltet hat.

Ignoriert man diesen Fall, so wird der Thread bei der erneuten Aktivierung der Domain "zerstört". Das heißt, der Threadkontrollblock wird für den erneuten Aufruf der Methode activated vorbereitet. Dadurch wird der Threadkontext, der bei der Unterbrechung des Schedulers gesichert wurde, verändert und kann somit nicht mehr restauriert werden. Die Folge ist, dass die Ausführung der Methode activated an einer beliebiger Stelle abgebrochen wurde. Die Daten des Schedulers könnten deshalb inkonsistent sein.

[9] Diese Anforderung muss vom Bytecode-Verifier überprüft werden.

Eine mögliche Lösung des Problems wäre die Forderung nach einer Obergrenze für die Laufzeit des Schedulers. Kann die Länge der Zeitscheiben jedoch dynamisch verändert werden, so ist das keine akzeptable Lösung.

Sinnvoller ist, dem Scheduler mitzuteilen, dass die Methode activated unterbrochen wurde. Dadurch können dann entsprechende Maßnahmen getroffen werden, um die Daten konsistent zu halten. Vom Basisinterface HighlevelScheduler werden deshalb die beiden Methoden Scheduler_preempted und Scheduler_interrupted gefordert. Diese werden anstatt der Methoden preempted bzw. interrupted aufgerufen, wenn statt eines normalen Threads der Schedulingthread unterbrochen wurde. Wie alle Informationsmethoden werden sie unter Interruptausschluss ausgeführt. Durch den Rückgabewert dieser Methoden kann der Scheduler dem System auch mitteilen, dass der unterbrochene Schedulingthread einfach fortgesetzt werden soll, anstatt neu mit der Ausführung der activated-Methode zu beginnen (siehe auch Anhang A.2).

Speicherverbrauch für Hilfsthreads

Für jeden Aktivitätsträger wird neben dem Threadkontrollblock auch Speicherplatz für den Stack reserviert.[10] Da zu jedem Highlevel-Scheduler zwei Threads erzeugt werden, ist der Speicherverbrauch relativ hoch. Die besondere Verwendung der Aktivitätsträger könnte jedoch dazu genutzt werden den Speicherverbrauch zu reduzieren.

Jedes Highlevel-Schedulerobjekt hat einen Thread, welcher nur Methoden unter Interruptausschluss ausführt. Dadurch ist sichergestellt, dass niemals zwei dieser Hilfsthreads – von verschiedenen Schedulern – auf einem Prozessor gleichzeitig aktiv sein können.

Es bietet sich deshalb an, allen Threads dieses Typs den gleichen Speicherbereich als Stack zuzuteilen. Somit müsste im gesamten System nur ein Stack für all diese Hilfsthreads zur Verfügung stehen. In Multiprozessorsystemen muss allerdings für jeden Prozessor ein solcher Speicherbereich eingerichtet werden, da auf unterschiedlichen Prozessoren gleichzeitig Scheduler aktiv sein können.

Schnittstellen von Schedulerportalen

Abbildung 5.8 zeigt detailliert den Ablauf des in der Einführung (Abbildung 5.5) dargestellten Domainwechsels. Man sieht, dass die Kommunikation der Scheduler nicht direkt abläuft, sondern Hilfsmethoden der Domain Zero genutzt werden müssen. Das kommt davon, dass ein Methodenaufruf über Domaingrenzen hinweg – d. h. mit einem Threadwechsel – nicht in Java realisierbar ist. Die verwendeten Methoden der Domain Zero werden allerdings nicht über normale Portale angesprochen (wie Abschnitt 5.5.4 zeigt, ist das auch gar nicht möglich). Stattdessen werden so genannte *inlined* Portale (siehe Abschnitt 2.2) verwendet. Der Code der Hilfsfunktionen wird somit direkt vom aufrufenden Thread ausgeführt, sodass kein Zeitverlust entsteht. Verglichen mit einem virtuellen Methodenaufruf an einem Java-Objekt ist ein *inlined* Portal sogar etwas schneller wie Tabelle 5.5 zeigt.

[10] Die Größe ist konfigurationsabhängig (z. B. 4KByte).

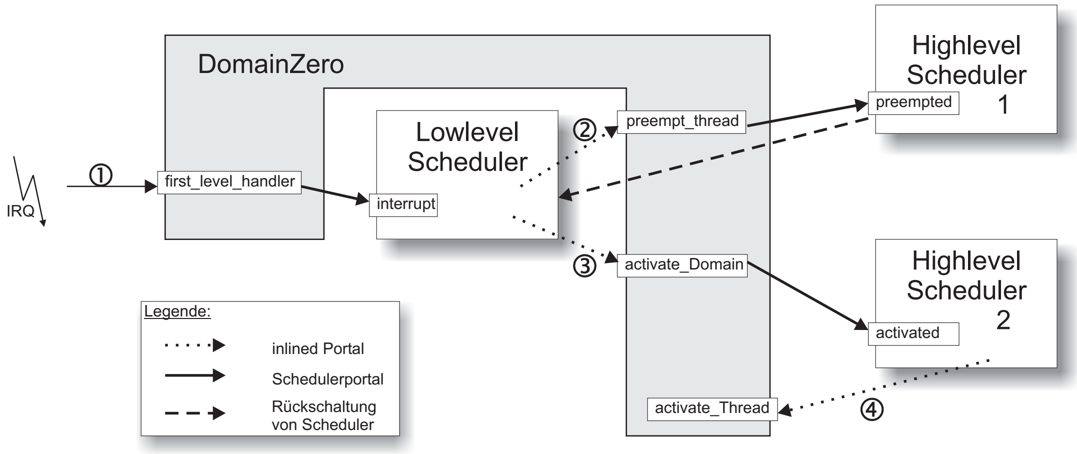


Abbildung 5.8: Kommunikation zwischen den Schedulern im Detail

	Zeit
Methodenaufruf	88 ns
inlined Portal	38 ns

(gemessen auf einem 500MHz Pentium III System)

Tabelle 5.5: Geschwindigkeitsvergleich zwischen Methodenaufruf und *inlined* Portal

Um die Kommunikation zwischen Low- und Highlevel-Scheduler komfortabler zu gestalten, könnte man ein Stellvertreter-Prinzip (Stub-Konzept) einrichten, wie es bei Portalen der Fall ist. Zu jedem Highlevel-Scheduler hätte der Lowlevel-Scheduler dann ein Stellvertreterobjekt, an dem er die entsprechenden Methoden aufrufen kann. Der Umweg über die Domain Zero würde somit transparent erfolgen. Ein Stellvertreter-Konzept könnte jedoch einen negativen Einfluss auf die Geschwindigkeit haben.

5.5.8 Alternative Interaktion mit Hilfe von Ereignislisten

Abbildung 5.9 zeigt die beteiligten Threads bei einem Domainwechsel. Zu Beginn ist Thread A aus Domain 1 aktiv. Dieser wird durch einen Timer-Interrupt unterbrochen. Daraufhin wird der Interrupthandler in einem Thread des Lowlevel-Schedulers ausgeführt. Dieser ruft zuerst die Methode `preempted` des Highlevel-Schedulers von Domain 1 auf, um die Domain darüber zu informieren, dass sie verdrängt wurde. Wie schon beschrieben, ist das mit einem Threadwechsel verbunden.

Anschließend aktiviert der Lowlevel-Scheduler die Domain 2. Dies geschieht durch Aufruf der Methode `activated` am Highlevel-Scheduler der neuen Domain, was ebenfalls mit einem Threadwechsel verbunden ist. Der aktivierte Scheduler schaltet letztendlich auf den neuen Thread B um.

Wie deutlich wird, ist ein Domainwechsel relativ aufwändig. Eine alternative Realisierung wäre die Einrichtung einer Ereignisliste für jeden Highlevel-Scheduler. In diese Liste werden alle

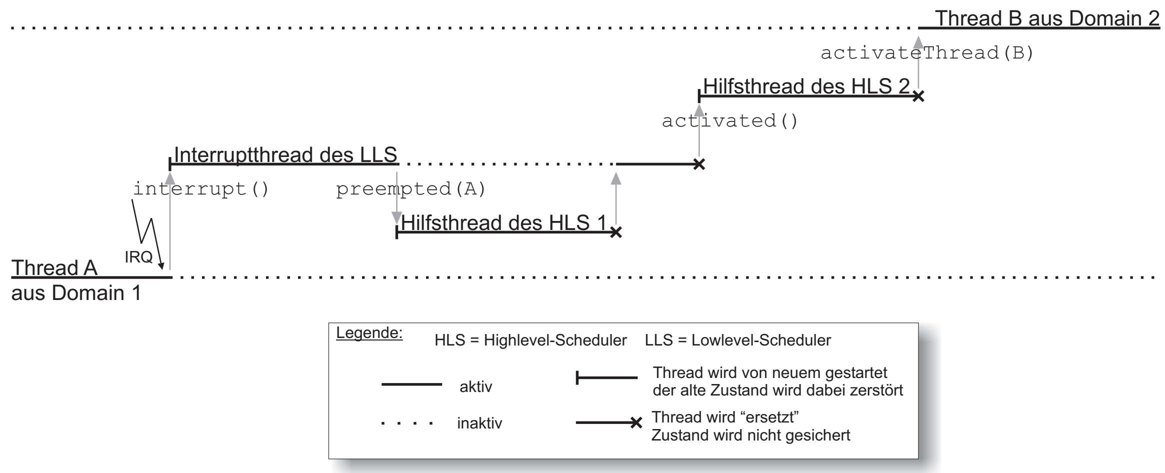


Abbildung 5.9: Beteiligte Threads bei einem Domainwechsel

auftretenden Ereignisse eingehängt, die "seine" Threads betreffen. Wird die entsprechende Domain aktiviert, so kann der Highlevel-Scheduler zuerst die Liste der Ereignisse durchgehen, um seine Informationsstrukturen zu aktualisieren. Anschließend kann er einen Thread aktivieren. Abbildung 5.10 zeigt, dass dadurch zwei Threadwechsel eingespart werden könnte.

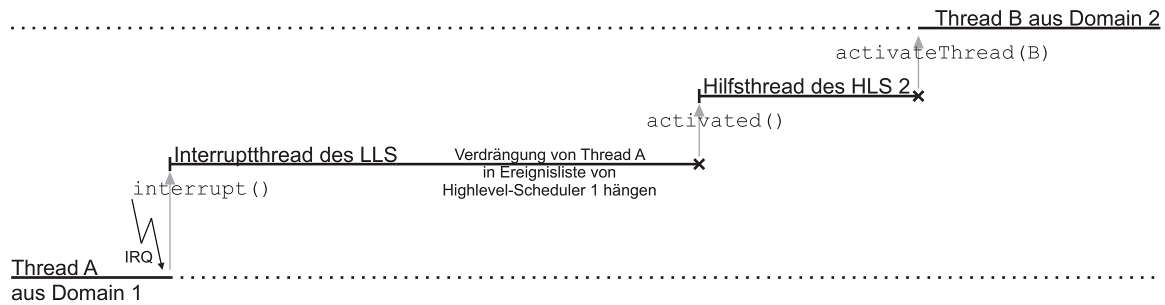


Abbildung 5.10: Ablauf bei alternativer Kommunikation am Beispiel eines Domainwechsels

Dieses Verfahren hat allerdings den Nachteil, dass die Ereignisse den Domains bzw. deren Schedulern erst bekannt werden, wenn die Domain wieder Rechenzeit bekommt. Speziell für Echtzeitbedingungen, ist dies aber nicht ausreichend.

5.6 Timer

Um die Rechenzeit zwischen mehreren Aktivitätsträgern aufzuteilen, ist es nötig, einem Thread nach einer vordefinierten Zeit den Prozessor zu entziehen. Dieses Prinzip wird oft durch eine Zeitscheibe abstrahiert, auf der jeder Aktivitätsträger einen Teil der Rechenzeit erhält. Ist seine Zeit abgelaufen, so wird er verdrängt und der nächste Thread aktiviert.

Die folgenden Abschnitte beschreiben einige Aspekte, die bei der Realisierung eines solchen preemptiven Konzepts in JX besonders beachtet wurden.

5.6.1 Zielsetzung

Für eine möglichst flexible Verwaltung der Rechenzeit werden folgende Punkte gefordert:

- In einem Multiprozessorsystem soll es möglich sein für die einzelnen CPUs unabhängig voneinander Zeitscheiben festzulegen.
- Die Zuteilung der Rechenzeit zu den Domains soll keinesfalls gleichmäßig erfolgen müssen. Der Lowlevel-Scheduler soll die Kontrolle darüber haben, ob er einer Domain besonders viel Rechenzeit überlässt, wohingegen eine andere Domain den Prozessor kaum nutzen darf.
- Jede Domain soll die ihr zugeteilte Rechenzeit individuell auf ihre Threads verteilen können. Das heißt, dass jeder Domain die Möglichkeit gegeben werden soll, eigene Zeitscheiben – innerhalb ihrer Rechenzeit – zu realisieren.

5.6.2 Timer in Multiprozessorsystemen

Um eine Zeitscheibe zu realisieren, wird ein Timer benötigt, sodass nach einer vordefinierten Zeit dem aktuellen Thread der Prozessor entzogen werden kann. Anschließend wird dem Scheduler ermöglicht, die Rechenzeit einem anderen Aktivitätsträger zuzuteilen.

In einem PC/AT-kompatiblen Monoprozessorsystem gibt es zwei mögliche Zeitgeber, die dafür verwendet werden können [Mes98]:

- den programmierbare Intervalltimer oder
- die Real-Time-Clock.

Beide stehen auch in einem Multiprozessorsystem zur Verfügung. Allerdings haben sie den Nachteil, dass sie nur einmal im System vorhanden sind. Die Forderung nach unabhängigen und individuellen Zeitscheiben pro CPU ist deshalb mit diesen Timern nur schwer zu erfüllen.

Prinzipiell wäre folgende – sehr aufwändige – Realisierung denkbar: eine zentrale Timerverwaltung wird über die gewünschten Intervalle aller CPUs informiert. Der Timer wird dann immer auf das kleinste verbleibende Intervall eingestellt. Ein auftretender Timer-Interrupt muss nun an den entsprechenden Prozessor weitergeleitet werden.

Effizienter ist die Verwendung der in Abschnitt 4.2.3 vorgestellten lokalen Timer. In einem Intel-MP-System hat jeder Prozessor durch seinen lokalen APIC einen Zeitgeber, der für jede CPU individuell programmiert werden kann. Eine zentrale Steuerung für alle CPUs ist dadurch nicht notwendig; die individuelle Festlegung unabhängiger Zeitscheiben ist möglich.

5.6.3 Globale Zeitscheibe

Wie in Abschnitt 5.4.1 bereits erwähnt, ist der Lowlevel-Scheduler für die Aufteilung der Rechenzeit zwischen den Domains verantwortlich. Er stellt deshalb einen Interrupthandler zur Verfügung, der bei einem Timer-Interrupt aktiviert wird.

Um die Zeitspanne für jede Domain individuell festlegen zu können, wird der Timer nicht im periodischen Modus verwendet, sondern im so genannten "one-shot-mode".[11] Das bedeutet, dass nach jedem Interrupt ein neues Intervall programmiert werden muss.

Der Overhead, der durch das Einstellen des nächsten Intervalls entsteht, liegt in der Größenordnung von etwa 160 ns, dies entspricht nur dem doppelten eines Methodenaufrufs, wie Tabelle 5.6 belegt. Durch die Verwendung eines speziellen *Compiler-Plug-In*s – wie in [Waw01] beschrieben – könnte dieser Overhead sogar noch verringert werden.

	Zeit
Methodenaufruf	88 ns
setTimer	157 ns

(gemessen auf einem 500MHz Pentium III System)

Tabelle 5.6: Zeitbedarf für das Einstellen eines neuen Timer-Intervalls

Der Vorteil dieses Timermodus liegt darin, dass die Intervalle nicht gleichlang sein müssen. Da die Länge des folgenden Intervalls ohnehin jedesmal neu festgelegt werden muss, ist eine individuelle Zeitverteilung leicht zu realisieren.

5.6.4 Individuelle Zeitscheibe pro Domain

Um die Verwaltung der Rechenzeit flexibel zu gestalten, muss für die Highlevel-Scheduler die Möglichkeit geschaffen werden, eigene Zeitscheibenstrategien zu implementieren. Sie müssen deshalb in der Lage sein, selbst ein Zeitintervall festzulegen, währenddessen ein Thread den Prozessor belegen darf. Die maximal erlaubte Rechenzeit für die Domain darf (normalerweise) jedoch nicht überschritten werden.

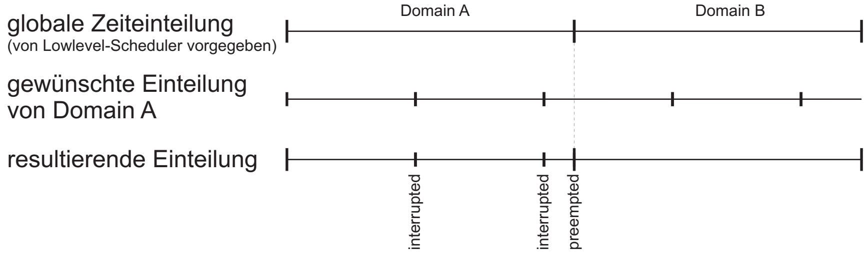


Abbildung 5.11: Konflikt zwischen globaler und domainspezifischer Zeitscheibe

[11]Dies gilt nur für den lokalen Timer im Multiprozessorbetrieb. Siehe dazu auch Abschnitt 5.6.7.

Ein gewünschtes Verhalten ist in Abbildung 5.11 dargestellt: Die Intervalllänge, die Domain A wünscht, ist kein Teiler der zur Verfügung stehenden Rechenzeit. Die letzte Aktivierung der Domain darf jedoch nicht länger sein als die verbleibende Rechenzeit.

Um die Zeit die eine Domain den Prozessor belegen darf einzuhalten, liegt die Verwaltung der individuellen Zeitscheiben im Verantwortungsbereich des Lowlevel-Schedulers. Ihm meldet ein Highlevel-Scheduler die gewünschte Intervalllänge. Aktiviert der Lowlevel-Scheduler eine Domain, so muss er feststellen, ob die verbleibende Rechenzeit noch ausreicht, um der Domain ein vollständiges Intervall zu geben. Ist das nicht der Fall, kann entweder die restliche Zeit zugeteilt (wie in Abbildung 5.11) oder gleich zur nächsten Domain gewechselt werden.

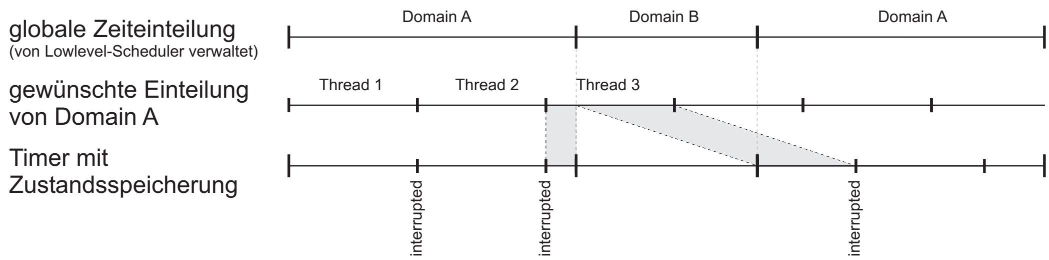


Abbildung 5.12: Timer mit Zustandssicherung

Der Lowlevel-Scheduler könnte sogar das in [Rie94] als "hierarchischer Timer mit Zustandsspeicherung" bezeichnete Konzept realisieren (siehe auch [RK96]). Dabei wird bei der nächsten Aktivierung der Domain die Kontrolle nicht an den Highlevel-Scheduler übergeben, sondern sofort der unterbrochene Thread aktiviert. Das Zeitintervall wird dabei auf die verbleibende Zeit eingestellt, sodass die gesamte Rechenzeit des Threads einem vollständigen Intervall entspricht (siehe Abbildung 5.12). Die Verdrängung der Domain könnte somit für den Highlevel-Scheduler transparent ablaufen, wenn das erwünscht ist.

Wie aus den gezeigten Beispielen hervorgeht, wird die Zeitscheibenstrategie durch die Kombination aus Low- und Highlevel-Scheduler realisiert. Unter bestimmten Rahmenbedingungen ist durch dieses Konzept jede Domain in der Lage, die ihr zur Verfügung stehende Rechenzeit individuell zu verwalten.

5.6.5 Benachrichtigungen

Um einem Highlevel-Scheduler den Grund einer Unterbrechung mitzuteilen, sind zwei verschiedene Benachrichtigungsmethoden vorgesehen (vgl. auch Abbildung 5.11):

preempted: zeigt an, dass die Rechenzeit der Domain verbraucht ist. Dem Aktivitätsträger wurde deshalb der Prozessor entzogen. Der Lowlevel-Scheduler wird daraufhin eine andere Domain aktivieren.

interrupted: weist lediglich darauf hin, dass die vom Highlevel-Scheduler festgelegte Zeitschranke überschritten wurde. Der aktuelle Thread wurde zwar verdrängt, die Domain darf den Prozessor jedoch weiterhin verwenden. Der Lowlevel-Scheduler wird deshalb dieselbe Domain sofort wieder aktivieren.

5.6.6 Bewertung

Die Verwaltung der Zeitscheiben durch den Lowlevel-Scheduler hat zur Folge, dass jeder Timer-Interrupt zuerst den Lowlevel-Scheduler aktiviert, egal ob die globale oder die individuelle Zeiteinteilung einer Domain für den IRQ verantwortlich ist. Im Interrupthandler wird anschließend der Grund des Interrupts geklärt, um die entsprechende Domain zu aktivieren. Sinnvoller wäre Interrupts zur Realisierung einer individuellen Zeitscheibe direkt an die entsprechende Domain zu leiten. Dabei muss jedoch beachtet werden, dass die globale Zeitscheibe nicht verletzt wird.

Statt den Lowlevel-Scheduler zu aktivieren könnte, die Einhaltung der globalen Zeitscheibe bereits im Kern überprüft werden, bevor der Interrupthandler eines Schedulers aktiviert wird. Allerdings wird dann die Freiheit der Prozessorverwaltung stark eingeschränkt. Zum einen müsste der Kern über alle Zeitscheiben informiert werden, zum anderen wäre die Reaktion bei ungenügender Restrechenzeit im Kern festgelegt und könnte somit nur schwer ausgetauscht werden.

5.6.7 Ausblick

In Abschnitt 5.6.3 wurde beschrieben, dass der Timer im "one-shot-mode" betrieben wird. Das trifft jedoch nur im Multiprozessorbetrieb zu. In einer Monoprozessorumgebung wird statt der lokalen Timer (die dort nicht existieren) der Intervall-Timer verwendet, um Zeitscheiben zu realisieren. Dieser wird jedoch – aus historischen Gründen – im periodischen Modus betrieben, um eine globale Zeitvariable zu erhöhen. Diese Variable wird anschließend dazu verwendet, einen "one-shot-Timer" zu simulieren.

Um die Zeitsteuerung einheitlich zu gestalten, könnte man das Scheduler-Konzept so erweitern, dass nicht mehr ein im Kern festgelegter Timer, sondern stattdessen ein Timerobjekt verwendet wird. Ein solches Objekt kann dann für jeden Timer im System stehen. Das Konzept wird dadurch modularer und flexibler. Bei der Realisierung ist allerdings darauf zu achten, dass durch die Einführung der neuen Abstraktionsschicht nicht zu viel Geschwindigkeit verloren geht.

5.7 Bewertung

Das in diesem Kapitel vorgestellte Schedulingkonzept erfüllt die angestrebten Anforderungen. Die Realisierung der Schedulerobjekte in Java ist flexibel und erlaubt die Realisierung beliebiger Schedulingstrategien.

Durch die Aufteilung der Threadverwaltung in zwei Stufen wird die Möglichkeit geschaffen, jeder Domain die Verwaltung ihrer Threads selbst zu übertragen. Dabei muss den einzelnen Schedulern nicht vertraut werden, da kritische Anforderungen von einem Bytecode-Verifier überprüft werden.

Auch für die Unterstützung von Multiprozessorsystemen lässt das Konzept verschiedene Konfigurationen zu. Der Implementierung ist die Freiheit gegeben, ob für jeden Prozessor ein eigener Scheduler eingerichtet wird oder ob ein Scheduler mehrere CPUs verwalten soll.

Insgesamt geht die Flexibilität allerdings zu Lasten der Ausführungsgeschwindigkeit, wie Tabelle 5.3 zeigt. Dabei geht die meiste Zeit durch die relativ aufwändige Benachrichtigung eines Highlevel-Schedulers verloren.

Durch die Deaktivierung einzelner nicht benötigter Benachrichtigungen kann man jedoch einen Kompromiss zwischen Geschwindigkeit und Grad der Information schließen. Eine allgemeine Verbesserung der Geschwindigkeit könnte durch die Einführung von Ereignislisten, wie sie in Abschnitt 5.5.8 beschrieben wurden, erreicht werden. Die damit verbundene Einschränkung bezüglich des Benachrichtigungszeitpunkts, könnte in vielen Fällen in Kauf genommen werden.

5.8 Zusammenfassung

In diesem Kapitel wurde ein zweistufiges Schedulerkonzept vorgestellt, um den Domains eine individuelle Verwaltung ihrer Rechenzeit zu ermöglichen. Die Verwaltung der Aktivitätsträger einer Domain wird jeweils durch ein eigenes Schedulerobjekt durchgeführt. Dieses Schedulerobjekt muss dabei nicht in der betroffenen Domain liegen. Die Auslagerung der Threadverwaltung in eine andere Domain ermöglicht ein dynamisches Rekonfigurieren des Systems.

Zur Verwaltung der Threads kann jede Domain eine Zeitscheibe einrichten, die ihre Rechenzeit in individuelle Intervalle fragmentiert. Bezüglich der Verwaltung ihrer Rechenzeit sind die Domains somit voneinander entkoppelt.

Auch die Unterstützung vom Multiprozessorsystemen ist flexibel gelöst. Man kann entweder für jeden Prozessor einen eigenen Scheduler installieren oder ein Schedulerobjekt für mehrere CPUs verwenden.

6 Portale

Dieses Kapitel beschäftigt sich mit der, im Rahmen dieser Arbeit vorgenommenen, Veränderung des Portalmechanismus. Zum einen wird gezeigt, dass der Scheduler mit einbezogen werden muss, wenn ein Portalaufruf stattfindet. Zum anderen wird auf Besonderheiten eingegangen, die im Zusammenhang mit Multiprozessorsystemen beachtet werden müssen.

6.1 Einführung

In Mikrokern-Betriebssystemen werden viele Dienste auf Benutzerebene bereitgestellt. In JX, welches zu dieser Gruppe von Betriebssystemen zählt, kann beispielsweise das Datei- oder das Netzwerksystem als eigenständige Domain realisiert werden.

Dieses modulare Konzept impliziert, dass eine Kommunikation zwischen den Komponenten – in JX *Domains* – möglich sein muss. Ein grundlegendes Konzept von JX ist jedoch die Entkopplung der Domains. Der direkte Aufruf einer Methode von einer anderen Domain aus würde dieses Konzept gefährden. Deshalb werden domainübergreifende Methodenaufrufe durch so genannte Portale realisiert.

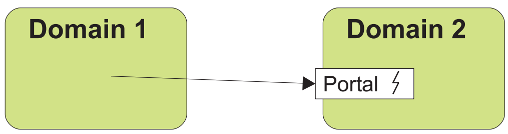


Abbildung 6.1: Portalaufruf

Jede Domain kann ausgewählte Methoden, die andere Domains aufrufen dürfen, mit Hilfe eines Portals bereitstellen. Um die vollständige Entkopplung der Domains sicherzustellen, wird mit jedem Portal ein Aktivitätsträger verknüpft. Dieser Thread hat nur die Aufgabe Methoden, die durch das Portal aufgerufen werden, auszuführen.

Ist das Portal beim System registriert, kann eine andere Domain (z. B. mit Hilfe eines Namensservices) eine Referenz auf das Portal erhalten. Dabei handelt es sich jedoch nicht wirklich um eine Referenz, sondern um ein Stellvertreterobjekt. An dem Objekt können die ausgewählten Methoden des Dienstanbieters aufgerufen werden. Das System sorgt daraufhin automatisch für eine Threadumschaltung zur Zieldomain.

Zusammenfassend ist festzuhalten, dass Portale im Allgemeinen der Interprozesskommunikation dienen und teilweise mit Systemaufrufen vergleichbar sind.

6.2 Ausgangssituation

Abbildung 6.2 zeigt eine Darstellung des Portalmechanismus zu Beginn dieser Arbeit:

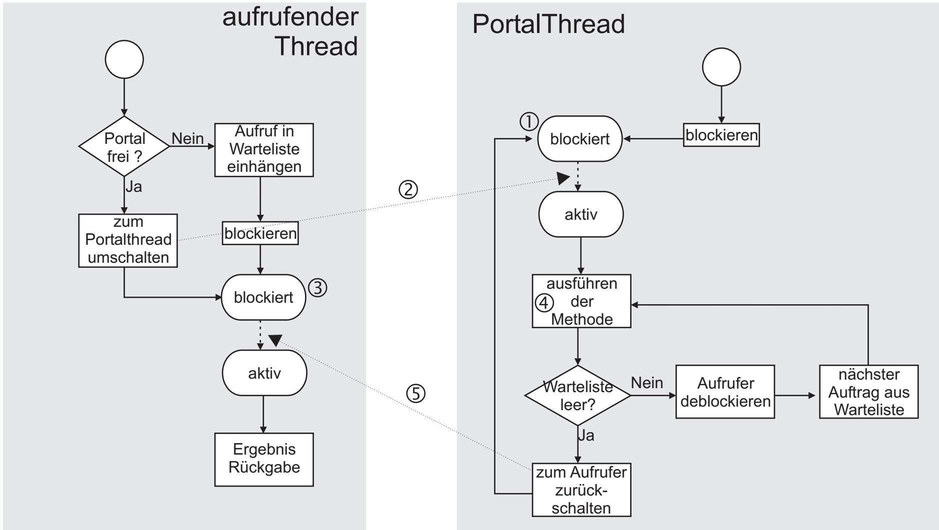


Abbildung 6.2: Portalmechanismus (vor dieser Arbeit)

- In der Ausgangsstellung ist der Portalthread, der die Methoden des Portals ausführt, blockiert (1).
- Ein Methodenaufruf an dem Portal führt zu einer sofortigen Threadumschaltung zum Aktivitätsträger des Portals (2).
- Der aktivierte Portalthread führt die Methode aus (4), während der Aufrufer auf das Ergebnis wartet (3).
- Nach Ausführung der Methode schaltet der Aktivitätsträger des Portals zurück zum Aufrufer (5) und blockiert sich wieder.

Da für das Rendezvous-Prinzip nur ein Portalthread zur Verfügung steht, werden Aufrufe die eintreffen während das Portal "besetzt" ist in eine Warteliste gehängt. Somit werden die Aufrufe an dem Portal sequentialisiert.

Der Warteschlangenmechanismus ist in Abbildung 6.2 dargestellt. Im Folgenden soll jedoch nur die Situation betrachtet werden, dass das Portal "frei" ist.

6.3 Motivation

Eine unerwünschte Eigenschaft der bestehenden Implementierung war, dass die Domains keinen Einfluss auf die Threadumschaltung hatten, die bei einem Portalaufruf stattfindet. Zwei

Beispiele sollen die Notwendigkeit belegen, auf den Portalmechanismus Einfluss nehmen zu können.

6.3.1 Kooperatives Scheduling

Zuerst soll die Situation einer Domain betrachtet werden, die eine kooperative Schedulingstrategie einsetzt.

Kooperatives Scheduling kann als das Gegenteil von preemptivem Scheduling angesehen werden: ein Thread wird nicht verdrängt, sondern gibt seine Rechenzeit immer wieder freiwillig ab, um den Prozessor einem anderen Aktivitätsträger zu überlassen. Vorteil dieses Verfahrens ist, dass ein Thread nicht an beliebiger Stelle unterbrochen wird, sondern nur an bestimmten, vordefinierten Punkten.

In Bezug auf eine Domain bedeutet das, dass ein Thread nicht unterbrochen wird, um einem anderen Thread derselben Domain den Prozessor abzugeben. Der Lowlevel-Scheduler kann trotzdem jederzeit die Domain – und damit auch den aktuellen Thread – verdrängen, um eine andere Domain zu aktivieren.

Speziell beim Entwurf eines Dienstes, der für die Verwendung durch mehrere Threads geeignet sein soll, kann sich der Programmieraufwand durch den Einsatz einer kooperativen Schedulingstrategie erheblich reduzieren. Hier könnte unter Umständen auf die Verwendung von Sperren verzichtet werden, indem nur an unkritischen Stellen der Prozessor aufgegeben wird. Es können somit Programmabschnitte festgelegt werden, die niemals gleichzeitig aktiv sind.

Als Beispiel einer Domain, die auf ein kooperatives Scheduling angewiesen ist, ist das Dateisystem von JX zu nennen. Für jede geöffnete Datei stellt das Dateisystem ein Portal zur Verfügung. An diesem Portal können Lese- oder Schreibanforderungen an das Dateisystem übermittelt werden.

Die Implementierung des Dateisystems ist an Linux angelehnt. Dort wird der Zugriff auf die Festplatte im Kern ausgeführt, wobei der Linuxkern Ununterbrechbarkeit garantiert. Die Realisierung ist deshalb so angelegt, dass neue Aufträge nur an bestimmten Stellen des Codes eintreffen dürfen. Wie oben beschrieben, führt ein Portalaufruf jedoch sofort zur Aktivierung des Portalthreads. Dadurch kann nicht gewährleistet werden, dass Aufträge an den vorgesehenen Stellen eintreffen.

Um ein kooperatives Scheduling zu realisieren, muss die Domain deshalb Kontrolle über die Aktivierung der Portalthreads haben.

6.3.2 Blockierende Portalmethoden

Das zweite Beispiel zeigt, dass es auch für den Aufrufer sinnvoll ist, Kontrolle über die Threadumschaltung zu haben:

In der beschriebenen Implementierung erfolgt bei einem Portalaufruf sofort ein Threadwechsel zur Zieldomain. Dadurch wird die verbleibende Rechenzeit an den Aktivitätsträger des Portals abgegeben. Grundsätzlich ist dieses Verhalten sinnvoll, da die Abarbeitung kleiner Methoden sofort angestoßen wird und daher auch schnell ein Ergebnis vorliegt.

Allerdings kann man sich auch Situationen vorstellen, wo ein solches Verhalten unerwünscht ist. Beispielsweise, wenn die aufgerufene Methode einen Zugriff auf ein langsames E-/A-Gerät beinhaltet. Der aufrufende Thread wird dadurch unter Umständen sehr lange blockiert.

Es wäre stattdessen sinnvoller, dass die Domain auf einen anderen Thread umschalten kann, sobald der aktuelle Thread einen Portalaufruf tätigt. Die Abarbeitung des Portals würde dann in der Zeitscheibe des Dienstanbieters stattfinden. Die bestehende Realisierung lässt dies nicht zu, da ein Portalaufruf immer sofort ausgeführt wird, wenn das entsprechende Portal "frei" ist.

6.4 Ziele

Die oben genannten Beispiele begründen die Forderung, dass eine Domain mehr Kontrolle über den Portalmechanismus haben soll. Außerdem müssen Multiprozessorsysteme bei Portalaufrufen besonders berücksichtigt werden. Folgende Ziele werden deshalb formuliert:

- Vollständige Integration des Schedulers in den Portalmechanismus und damit
    - vollständige Kontrolle des Portalnutzers und
    - vollständige Kontrolle des Portalanbieters.
- Unterstützung von Multiprozessorsystemen durch:
    - Berücksichtigung der CPU-Beschränkung bei Umschaltung auf einen Portalthread.
    - Simultane Nutzung eines Portals von mehreren CPUs.

6.5 Integration des Schedulers

6.5.1 Kontrolle des Portalnutzers

Wie bisher bereits deutlich gemacht wurde, ist es wünschenswert, dass die Domain, die ein Portal nutzt, entscheiden kann, ob die verbleibende Rechenzeit dem Portalaufruf überlassen werden soll oder nicht.

Diese Entscheidung könnte bei jedem Aufruf eines Portals getroffen werden. Um die Wahl mitzuteilen müssten die Methoden des Stellvertreters allerdings um einen Parameter erweitert werden. Das ist jedoch unerwünscht, da der Stellvertreter dann andere Schnittstellen bereitstellt als das Originalobjekt. Folglich wären sie nicht mehr austauschbar ist.

Alternativ könnte man bereits bei der Erstellung eines Portalstellvertreters festlegen, wie Aufrufe durchzuführen sind. Dann kann jedoch nicht mehr für jeden Aufruf individuell entschieden werden.

Da die Verwaltung der Rechenzeit in den Aufgabenbereich des Highlevel-Schedulers einer Domain fällt, kann man die Entscheidung auch dem Scheduler übertragen. Diese Möglichkeit wurde realisiert. Um dem Scheduler über den bevorstehenden Portalaufruf zu informieren, wurden zwei Benachrichtigungsmethoden eingeführt:

- Die Methode blockedInPortalCall, meldet einen Thread, der einen Methodenaufruf an einem Portal gestartet hat. Der Portalthread war allerdings nicht in Warteposition, sodass der Aufruf in die Warteliste eingehängt wurde. Eine Übertragung der verbleibenden Rechenzeit auf den Portalthread ist daher nicht sinnvoll und auch nicht vorgesehen.

- Wenn das Portal "frei" ist, so wird stattdessen die Methode portalCalled aufgerufen. Diese Methode signalisiert dem Scheduler, dass sofort auf den Portalthread gewechselt werden könnte. Der Scheduler teilt durch einen entsprechenden Rückgabewert mit, ob die verbleibende Rechenzeit an die Domain des Portals abgegeben werden soll.

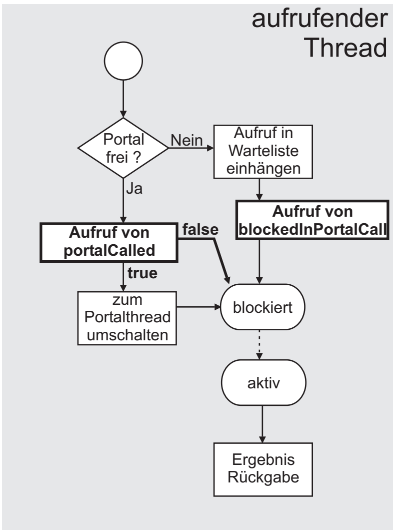


Abbildung 6.3: Portalmechanismus mit Benachrichtigung des Schedulers einer aufrufenden Domain

Entscheidet sich eine Domain dafür, die Rechenzeit abzugeben, so kann sie sich jedoch nicht sicher sein, dass der Dienstanbieter die Rechenzeit auch für die Ausführung der Methode nutzt. Selbst wenn der Anbieter nicht fehlerhaft oder böswillig implementiert ist, kann er unter Umständen den Methodenaufruf momentan nicht zulassen (vgl. Beispiel in Abschnitt 6.3.1).

Die Erweiterung des Portalmechanismus ist in Abbildung 6.3 dargestellt. Der folgende Abschnitt geht auf die möglichen Reaktionen eines Portalanbieters ein.

6.5.2 Kontrolle des Dienstanbieters

Aufgabe des Highlevel-Schedulers einer Domain ist, die Aktivitätsträger zu verwalten. Darunter fällt auch die Kontrolle, welche Threads aktiv sind und welche nicht. Wird der Aktivitätsträger eines Portals jedoch implizit – durch einen Portalaufruf – aktiviert, so wird der Scheduler umgangen. Wie das Beispiel in Abschnitt 6.3.1 zeigt, können dann bestimmte Eigenschaften, wie zum Beispiel die des kooperativen Schedulings, nicht gewährleistet werden. Die Möglichkeiten eines Schedulers werden somit eingeschränkt.

Die Aktivierung eines Threads muss deshalb ausschließlich vom Scheduler vorgenommen werden. Um das zu gewährleisten, ist eine spezielle Methode des Highlevel-Schedulers vorgesehen: switchTo. Sie erwartet als einziges Argument einen Thread und wird vom Kern bzw. Portalmechanismus aufgerufen, wenn ein bestimmter Aktivitätsträger aktiviert werden soll.

Im Normallfall sollte der Scheduler dieser Aufforderung auch nachkommen, da die Domain einen Dienst zur Verfügung stellen möchte. Allerdings hat man durch diese Methode auch die Möglichkeit, ein Umschalten zu verhindern. Der Scheduler hat somit die Kontrolle darüber, welcher Thread aktiviert wird.

Soll zum Beispiel eine kooperative Schedulingstrategie implementiert werden, so wäre es sinnvoll zuerst den Thread zu aktivieren, der im kritischen Abschnitt verdrängt wurde (falls vorhanden). Erst wenn dieser einen definierten Zustand erreicht hat und den Prozessor abgibt, sollte der Portalthread aktiviert werden.

Der Nutzer eines Portals kann dadurch allerdings nicht mehr sicher sein, dass seine Rechenzeit – die er dem Portal überlassen hat – auch wirklich zur Bearbeitung des Methodenaufrufs verwendet wird. Er muss dem Portalanbieter vertrauen, dass er die Rechenzeit dafür einsetzt, den Dienst möglichst schnell zu erbringen.

6.6 Multiprozessorunterstützung

6.6.1 Einhaltung der Ressourcenbeschränkung

Um Portale auch in einer Multiprozessorumgebung nutzen zu können, sind zusätzliche Anforderungen zu beachten.

Wie in Abschnitt 3.4 erklärt, wird einer Domain bei ihrer Erzeugung das Recht mitgegeben, bestimmte Prozessoren zu nutzen, wohingegen andere nicht verwendet werden dürfen.

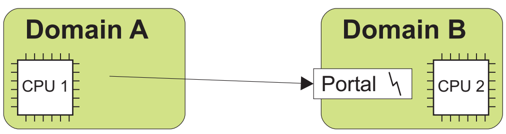


Abbildung 6.4: Portalaufruf mit inkompatiblen Prozessornutzungsrechten

Die Einschränkung auf bestimmte Prozessoren kann bei einem Portalaufruf zu Problemen führen. Als Beispiel soll die in Abbildung 6.4 schematisch dargestellte Situation dienen: Domain A darf nur Prozessor 1 verwenden; Domain B nur Prozessor 2. Nun möchte die Domain A einen Dienst der Domain B nutzen. B stellt zu diesem Zweck ein Portal zur Verfügung.

Ruft nun Domain A das Portal auf, so aktiviert ein Aktivitätsträger auf CPU 1 den Portalmechanismus. Dort muss festgestellt werden, dass die Zieldomain diesen Prozessor nicht verwenden darf. Eine direkte Umschaltung würde den Portalthread auf CPU 1 aktivieren. Da dieser Thread jedoch zu Domain B gehört, darf er nur auf CPU 2 ablaufen.

Abgesehen davon, dass durch den Portalthread die Ressourcenzuordnung missachtet werden würde, hätte die Domain B für den Prozessor 1 keinen Highlevel-Scheduler registriert. Der Kern könnte deshalb auch kein Schedulerobjekt beauftragen den Portalthread zu aktivieren.

Auf Multiprozessorrechnern darf die Rechenzeit somit nur an ein Portal abgegeben werden, wenn zum einen das Portal "frei" ist und zum anderen die CPU des Portalaufrufers vom Anbieter genutzt werden darf.

Im eben geschilderten Fall sieht die erstellte Implementierung vor, den Portalthread lediglich als lauffähig zu markieren, und das dem Highlevel-Scheduler der Domain B mitzuteilen. Hätte Domain B auch einen Scheduler für einen weiteren Prozessor (CPU 3) registriert, so würde derjenige Scheduler informiert, auf dessen Prozessor der Portalthread gerade läuft bzw. auf dem er zuletzt aktiv war.

6.6.2 Mehrere Threads pro Portal

In vielen Betriebssystemen darf im Multiprozessorbetrieb der Kern nicht von mehreren Prozessoren gleichzeitig betreten werden bzw. nur kleine Teile des Kerns sind für eine parallele Ausführung auf mehreren CPUs freigegeben. Somit kann ein Engpass entstehen, der die Performanz der gesamten Systems beeinflusst.

Für zentrale Komponenten könnte in JX dasselbe Problem auftreten. Als Beispiel soll ein HTTP-Server betrachtet werden, der einen Netzwerkdienst benötigt, um seine Aufgabe zu erfüllen. Ist das System modular aufgebaut, so wird eine Netzwerk-Domain diesen Dienst über ein Portal zur Verfügung stellen.

Wie in den vorherigen Abschnitten erwähnt, wird für das Portal ein Thread angelegt. An diesem Thread werden die Zugriffe auf die Methoden serialisiert. Sind nun mehrere Instanzen des HTTP-Servers aktiv, um mehrere Clients zu bedienen, so wird dieses Portal schnell zum Engpass.

Um das Problem zu mindern, könnte die Netzwerk-Domain die Funktionalität statt nur durch ein Portal, durch mehrere Portale zur Verfügung stellen. Jedes dieser Portale würde dann einen eigenen Aktivitätsträger besitzen. Die HTTP-Server müssten dann allerdings selbst entscheiden, welches der Portale sie nutzen. Da jedoch keine Informationen über die Auslastung der einzelnen Portale vorliegen, fehlt die Grundlage, auf der die Entscheidung getroffen werden soll.

Es ist daher wünschenswert, dass ein Portal von mehreren Aufrufern simultan genutzt werden kann. Um das zu erreichen, müssen einem Portal mehrere Aktivitätsträger zugeordnet werden.

Man möchte jedoch nicht von allen Diensten fordern, dass sie für die simultane Abarbeitung auf mehreren CPUs geeignet sein müssen. Deshalb muss die Domain, die ein Portal erzeugt, selbst entscheiden können, wie viele Aktivitätsträger sie mit einem Portal verknüpft.

Wird ein Portal aufgerufen, welches durch mehrere Portalthreads bearbeitet werden kann, so wird überprüft, ob noch ein wartender Thread vorhanden ist. Falls nicht, so ist das Portal "besetzt" und der Auftrag wird in die Auftragsliste eingehängt.

Ist hingegen ein Portalthread bereit, einen Auftrag auszuführen, so kann dieser sofort genutzt werden, vorausgesetzt, die aktuelle CPU ist für das Portal zulässig (siehe vorheriger Abschnitt).

Auch in einem Monoprozessorsystem kann es sinnvoll sein, einem Portal mehrere Threads zur Bearbeitung zuzuordnen. Für sehr kurze Methoden ist es oft wünschenswert, sie sofort auszuführen und Rechenzeit vom Aufrufer zur Verfügung zu stellen. Wenn das Portal nun "besetzt" ist, so ist das nicht möglich. Sind mehrere Aktivitätsträger mit einem Portal verbunden, so steigt dadurch die Wahrscheinlichkeit, dass das Portal "frei" ist.

6.7 Optimierung

Um die Ausführung eines Portals generell zu beschleunigen, wurde der Portalmechanismus (siehe Abbildung 6.2) dahingehend verändert, dass der Portalthread nach Beendigung der Methode nicht gesichert wird. Das heißt, durch die Umschaltung zum aufrufenden Thread (Punkt (5) in Abbildung 6.2) wird der Portalthread "zerstört". Bei einem neuen Aufruf (2), wird der Thread dann vorbereitet, um direkt die gewünschte Methode auszuführen. Diese Realisierung ist an die Kommunikation zwischen Schedulern angelehnt (siehe Abschnitt 5.5.4) und sollte einen erheblichen Geschwindigkeitsvorteil gegenüber der ursprünglichen Implementierung bringen, da der Portalthread nicht gesichert werden muss.

Es zeigte sich jedoch, dass das Vorbereiten des Portalthreads bei einem neuen Aufruf relativ viel Zeit in Anspruch nimmt. Tabelle 6.1 zeigt den Geschwindigkeitsgewinn, der durch die Optimierung erreicht wird. Die Messungen des erweiterten Portalmechanismus im nächsten Abschnitt (Tabelle 6.2) basieren auf dieser Optimierung.

	Portalaufruf
optimierter Portaltmechanismus	2,365 μs
ursprünglicher Portalmechanismus	2,430 μs

Tabelle 6.1: Geschwindigkeit des optimierten Portalmechanismus

6.8 Geschwindigkeitsvergleich

Portalaufruf mit unterschiedlicher Schedulerintegration

Tabelle 6.2 zeigt, wie die zusätzliche Interaktion mit den Schedulern die Ausführungsgeschwindigkeit eines Portals beeinträchtigt. Gemessen wurde die Zeit für einen Methodenaufruf über ein Portal bei verschiedenen Konfigurationen.

Benachrichtigung des Schedulers von	Portalmechanismus erweitert	Portalmechanismus vorher
Aufrufer und Anbieter	2,365 μs	—
nur Aufrufer (portalCalled)	1,922 μs	—
nur Anbieter (switchTO)	1,842 μs	—
ohne	1,362 μs	1,039 μs

(gemessen auf einem 500 MHz Pentium III System)

Tabelle 6.2: Zeit für einen Portalaufruf bei verschiedenen Konfigurationen

Die erste Zeile zeigt die Zeit für einen Aufruf, bei dem sowohl der Scheduler der aufrufenden Domain als auch der Scheduler des Portalanbieters mit einbezogen wurde – wie in diesem Kapitel beschrieben. Verglichen mit dem Portalmechanismus vor Beginn dieser Arbeit (letzte Spalte) wird deutlich, dass die erweiterte Funktionalität – die Interaktion mit den Highlevel-Schedulern –, relativ viel Zeit in Anspruch nimmt.

Da nicht jede Domain auf den Portalmechanismus Einfluss nehmen möchte, wurden auch Messungen durchgeführt, bei denen nur die aufrufende bzw. nur die anbietende Domain in den Portalaufruf mit einbezogen wurde. Dafür wurden Scheduler eingesetzt, welche die entsprechenden Benachrichtigungsmethoden nicht bereitstellen (siehe Abschnitt 5.5.6). Implementiert ein Scheduler eine der Benachrichtigungsmethoden nicht, so wird die Vorgehensweise des ursprünglichen Portalmechanismus ausgeführt.

Festzustellen ist, dass durch den Verzicht auf die Benachrichtigung eines Schedulers 440 μs bzw. 520 μs eingespart werden können. Diese Werte entsprechen etwa dem einer Threadumschaltung zu einem Scheduler (vgl. Tabelle 5.2).

Die letzte Zeile enthält das Ergebnis einer Messung, bei der weder der Aufrufer noch der Anbieter des Portals den Aufruf beeinflussen wollte. Diesen Wert kann man direkt mit dem ursprünglichen Portalmechanismus vergleichen.

Dabei ist festzustellen, dass bei gleicher Funktionalität (ohne Integration der Scheduler) die erweiterte Implementierung etwa 30 % mehr Zeit benötigt. Dies ist darauf zurückzuführen, dass der Highlevel-Scheduler zu einem bestimmten Thread gesucht werden muss, um anschließend festzustellen, ob er über das eingetroffene Ereignis informiert werden will. Bei einer Optimierung des Portalmechanismus sollte diesem Teil besondere Aufmerksamkeit geschenkt werden.

Portalausführung auf verschiedenen Prozessoren

Ausführung des Portals auf	Zeit
aktueller CPU	9,875 μs
anderer CPU	12,762 μs

Tabelle 6.3: Geschwindigkeitsvergleich zwischen einer Portalausführung auf der aktuellen CPU und auf einer anderen CPU

Tabelle 6.3 zeigt einen Geschwindigkeitsvergleich zwischen einem Portalaufruf, der von der aktuellen CPU und einem, der von einem anderen Prozessor bearbeitet wird. Bei beiden Messungen wurde der Domain des Portals keine Zeit zur Verfügung gestellt um das Portal sofort auszuführen. Der Portalthread wurde somit immer erst in der Zeitscheibe des Anbieters aktiviert.[1]

Festzustellen ist, dass die Ausführung auf einer anderen CPU deutlich länger braucht, als die Ausführung auf dem aktuellen Prozessor. Dies ist darauf zurückzuführen, dass die Zustandsänderungen einiger Aktivitätsträger (z. B. des Portalthreads) einer anderen CPU mittels Interprozessorinterrupt mitgeteilt werden müssen.

6.9 Ausblick

Es kommt häufig vor, dass eine Domain einen Dienst nutzt und auf dessen Ergebnis angewiesen ist. Beispielsweise kann ein HTTP-Server eine Anfrage üblicherweise erst beantworten, sobald er die entsprechenden Daten vom Dateisystem erhalten hat. Es ist daher wünschenswert, dass eine Domain die Möglichkeit hat, ihre Rechenzeit solange der Bearbeitung des Portals zur Verfügung zu stellen, bis das Ergebnis vorliegt.

Ein Grundprinzip von JX ist jedoch, dass die Ressourcen – hier im speziellen die Rechenzeit – an die Domains gebunden sind. Die bisherige Implementierung unterstützt daher die Möglichkeit, Rechenzeit an einen Portalaufruf zu binden, nicht ausreichend. Wie in diesem Kapitel beschrieben, kann lediglich die verbleibende Rechenzeit der aktuellen Zeitscheibe an die Domain eines Portals abgegeben werden. Wurde die Methode bis zum Ende der Zeitscheibe nicht abgeschlossen, so wird der Portalthread verdrängt. Nur der Anbieter des Portals kann den Portalthread dann wieder aktivieren (sobald er Rechenzeit hat). Eine erneute Aktivierung durch die Domain des Aufrufers ist nicht vorgesehen.

Folgende Ansätze sind möglich, um einem Portalaufruf über mehrere Zeitscheiben hinweg Rechenzeit abzugeben:

- Zum einen könnte die aufrufende Domain, sobald ihr vom Lowlevel-Scheduler wieder Rechenzeit zugeteilt wurde, diese explizit in die Bearbeitung des Portals investieren.

[1] Um eine Ausführung des Portals auf einem anderen Prozessor zu erreichen wurde der Domain des Portals die Nutzung der “aktuellen” CPU verboten. Eine sofortige Ausführung des Portals wäre somit nicht möglich gewesen (vgl Abschnitt 6.6.1).

Um das zu realisieren, könnte man im einfachsten Fall einen speziellen Systemaufruf vorsehen.
Vorteil dieses Verfahrens ist, dass es sich relativ leicht implementieren lässt. Demgegenüber steht, dass viel Zeit verbraucht wird, da zuerst die aufrufende Domain (vom Lowlevel-Scheduler) aktiviert wird, welche dann erst die Rechenzeit an den Portalthread abgibt. Um die Auswirkung zu verdeutlichen soll wieder das Beispiel des HTTP-Servers betrachtet werden: dieser fordert beim Dateisystem Daten an. Das Dateisystem wiederum überträgt die Aufgabe an den Festplattentreiber oder an den Netzwerkdienst, falls die angeforderte Datei auf einem anderen Rechner liegt.
Wird nun diese Aufrufkette unterbrochen und später fortgesetzt, so wird im schlechtesten Fall zuerst die Domain des HTTP-Servers aktiviert, diese gibt die Rechenzeit an das Dateisystems ab, dort wiederum wird die Rechenzeit an den Netzwerkdienst weitergegeben.
Durch die Verschachtelung wird ein Großteil der Rechenzeit bereits verbraucht, bevor die gewünschte Methode aktiviert wird.

- Eine anderer Ansatz ist, den Lowlevel-Scheduler als Ressourcen-Manager einzusetzen. Vor der Aktivierung einer Domain überprüft dieser, ob bei der vorherigen Verdrängung ein Portalthread aktiv war. Er kann somit automatisch statt der Domain den entsprechenden Portalthread aktivieren.
Vorteil diese Konzepts ist, dass bei Verkettungen von Portalaufrufen sofort das letzte Glied der Kette, aktiviert werden kann.
Zu beachten ist jedoch, dass die aufrufende Domain bei diesem Ansatz keinen eigenen Code mehr ausführen kann, bis der Portalaufruf zurückkehrt. Es muss deshalb eine Möglichkeit vorgesehen werden, den Portalaufruf abzubrechen, indem man beispielsweise eine Zeitschranke vorgibt.
Außerdem müssen Maßnahmen getroffen werden, wenn der Portalthread durch seine eigene Domain aktiviert und der Portalaufruf in dieser Zeitscheibe fertiggestellt wird. Der Lowlevel-Scheduler kann diesen Fall nicht selbständig erkennen, da er keine Informationen über die Threads einer Domain erhält. Er würde somit wieder den Portalthread aktivieren, sobald er der aufrufenden Domain Rechenzeit zuteilt.

Beide Ansätze haben ihre Vor- und Nachteile. Der erste Ansatz ist hauptsächlich dann geeignet, wenn nur kurze Portalaufrufe getätigt werden und diese auch kaum weitere Portale in Anspruch nehmen. Der zweite Ansatz hingegen ist vorzuziehen, wenn stark verkettete Portale üblich sind.

6.10 Zusammenfassung

Portale erlauben domainübergreifende Methodenaufrufe durch Threadumschaltung in die Zieldomain. Um kurze Methoden sofort auszuführen, kann der Aufrufer Rechenzeit an das Portal abgeben. Da hierbei die Verwaltung der Aktivitätsträger und der Rechenzeit beeinflusst wird, besteht die Möglichkeit, die Scheduler der entsprechenden Domains (Aufrufer- und Zieldomain) mit einzubeziehen.

Für Multiprozessorsysteme wurde außerdem eine Erweiterung beschrieben, die es erlaubt, dass mehrere Domains ein Portal simultan nutzen können.

7 Zusammenfassung und Ausblick

7.1 Zusammenfassung

Ziel der vorliegenden Arbeit war, das Java-Betriebssystem JX um eine Unterstützung für Intel-Multiprozessorsysteme zu erweitern. Dazu wurde die Steuerung der Interrupts sowie die Verwaltung der Aktivitätsträger angepasst. Im Rahmen dessen wurde auch ein flexibles Schedulingkonzept realisiert.

Um ein System mit mehreren Prozessoren leicht zu verwalten, werden alle Prozessoren durch CPU-Objekte abstrahiert. Ein Monoprozessorsystem unterscheidet sich somit nur in der Anzahl der CPU-Objekte von einem Multiprozessorsystem. Des Weiteren lässt diese Abstraktion eine objektorientierte Betrachtung zu. Durch die kontrollierte Weitergabe der CPU-Objekte lässt sich genau steuern, welche Anwendung oder welcher Dienst einen bestimmten Prozessor nutzen darf.

Die Verwaltung der Interrupts wurde an die Hardware eines Intel-Multiprozessorsystems angepasst. Es ist nun möglich, den Zielprozessor eines Interrupts frei auszuwählen. Zusätzlich kann für jeden Prozessor und jeden Interrupt ein eigener IRQ-Handler eingerichtet werden. Es gibt dadurch keinen Prozessor der eine Sonderstellung einnehmen muss; die Verwaltung von Hardwaregeräten kann auf beliebige CPUs verteilt werden.

Weiterhin wurde die Verwaltung der Aktivitätsträger verändert, um mehrere Prozessoren zu unterstützen. Dazu wurde ein zweistufiges Schedulingkonzept entwickelt und implementiert, welches eine individuelle Threadverwaltung der Domains unterstützt. So wird die verfügbare Rechenzeit primär nicht mehr unter den Aktivitätsträgern aufgeteilt, sondern unter den Domains. Diese wiederum können die ihnen zugeteilte Rechenzeit individuell ihren Threads zuteilen. Hierdurch ist es möglich, dass die Schedulingstrategie innerhalb einer Domain optimal an deren Bedürfnisse angepasst werden kann.

Da die Schedulerobjekte in Java realisiert sind, ist auch eine Kommunikation zwischen der Anwendung und ihrem Scheduler möglich. Dadurch kann sogar zur Laufzeit Einfluss auf die Verwaltung der Rechenzeit genommen werden, um die Schedulingstrategie den – eventuell veränderten – Gegebenheiten anzupassen.

Um die Schedulingstrategie anzupassen, unterstützt das Konzept auch den Austausch eines Schedulerobjekts zur Laufzeit. Dadurch können verschiedene modulare, separat entwickelte Scheduler verwendet werden, die im Laufe der Anwendung gewechselt werden. Zu diesem Zweck ist die Verwaltung der Threads durch eine Management-Domain vorgesehen.

Um Multiprozessorsysteme zu unterstützen, wird für jeden Prozessor, den eine Domain verwenden darf, ein separater Verweis auf ein Schedulerobjekt gespeichert. Diese Verweise müssen jedoch nicht auf unterschiedliche Objekte zeigen, sondern können auch das gleiche Objekt referenzieren. So kann für jede Domain einzeln entschieden werden, ob jeder Prozessor von einem eigenen Scheduler verwaltet oder ob für mehrere CPUs zusammen ein Schedulerobjekt eingesetzt werden soll. Lastverteilung kann somit durch eine gemeinsame Warteschlange oder mittels Kommunikation der einzelnen Scheduler erreicht werden.

Damit die Scheduler jederzeit die volle Kontrolle über die Aktivitätsträger einer Domain behalten, wurde auch der Portalmechanismus angepasst. Threadumschaltungen, die im Rahmen eines Portalaufrufs stattfinden, können nun sowohl von der aufrufenden, als auch durch die anbietende Domain beeinflusst werden.

7.2 Bewertung und Ausblick

Durch die Realisierung der Scheduler in Java und durch verschiedene Konfigurationsmöglichkeiten erreicht das Schedulingkonzept ein hohes Maß an Flexibilität. Dies geht allerdings zu Lasten der Ausführungsgeschwindigkeit. Dafür lässt das Konzept aber eine optimale Anpassung der Schedulingstrategie an die Anwendung zu. Der zusätzliche Zeitaufwand könnte dadurch teilweise kompensiert werden.

Da ein Großteil der Ausführungszeit für die Informationsübermittlung an die Schedulerobjekte verbraucht wird, ist der Ansatz, Benachrichtigungen beliebig zu deaktivieren (siehe Abschnitt 5.5.6) eine sinnvolle Konfigurationsmöglichkeit. Eine einfache Implementierung dieser Funktionalität zeigte allerdings eine geringere Wirkung als erhofft (siehe Tabelle 6.2). Die Überprüfung, ob ein Scheduler über den Zustandswechsel seines Threads informiert werden möchte, benötigt relativ viel Zeit. Eine Optimierung dieses Bereichs erscheint daher erfolgversprechend.

Einen alternativen Ansatz, Scheduler über Ereignisse zu informieren, zeigt Abschnitt 5.5.8. Die Ereignislisten, die dort skizziert werden, erlauben jedoch keinerlei Echtzeitbedingungen, da die Informationen über den Zustandswechsel eines Threads dem Scheduler nicht sofort zur Verfügung stehen.

Um den Dienst einer anderen Domain optimal zu nutzen, ist es – im Rahmen eines Portalaufrufs – möglich, dieser Domain die verbleibende Zeitscheibe zu überlassen. In Abschnitt 6.9 werden Möglichkeiten vorgeschlagen, dies auch über mehrere Zeitscheiben hinweg zu realisieren.

Auch im Bezug auf die Interruptverwaltung könnten Veränderungen vorgenommen werden. Zum Beispiel wird das in Abschnitt 4.1 kurz erwähnte Prioritätsregister eines Prozessors nicht genutzt. Dieses könnte jedoch zur Lastverteilung eingesetzt werden. Man sollte dem Benutzer deshalb eine Schnittstelle anbieten, die eine Nutzung des Registers und der daraus entstehenden Möglichkeiten zulässt.

Im Laufe dieser Arbeit ist nur am Rande auf die Bearbeitung eines Interrupts eingegangen worden. Es wurde festgestellt, dass ein IRQ-Handler installiert werden kann, der in einem eigenen Thread (ähnlich einem Portal) ausgeführt wird. Ob und in welchem Maße ein Scheduler auf diesen Aktivitätsträger Einfluss nehmen soll, müsste jedoch noch untersucht werden.

Abschließend ist festzuhalten, dass sich durch die angepasste Interruptverwaltung und das neue Schedulingkonzept die zur Verfügung stehenden Ressourcen gut auf die Threads bzw. Domains verteilen lassen. JX ist somit in der Lage Multiprozessorsysteme zu unterstützen.

A Kern-Schnittstellen

Dieser Anhang gibt einen Überblick über die Methoden eines Low- und eines Highlevel-Schedulers. Außerdem werden drei Portale der Domain Zero beschrieben. Zwei davon stellen Hilfmethoden für Scheduler bereit, das dritte gibt dem Benutzer die Möglichkeit Multiprozessorsysteme zu nutzen.

A.1 Lowlevel-Scheduler

A.1.1 Das Interface `LowlevelScheduler`

Ein Lowlevel-Schedulerobjekt wird dadurch ausgezeichnet, dass es das Interface `jx.zero.LowLevelScheduler` implementiert. Im Folgenden werden alle Methoden des Interfaces kurz beschrieben. Zu beachten ist, dass die Methoden ausschließlich vom Kern aufgerufen werden, wobei die Interrupts deaktiviert sind.

registered

```
public void registered(int irq_nr)
```

Diese Methode wird aufgerufen, nachdem der Lowlevel-Scheduler beim Kern registriert wurde. Der Parameter `irq_nr` ist die Nummer des verwendeten Timer-Interrupts (vgl. Abschnitt 5.6.7). Der Code dieser Methode sollte einen Interrupthandler registrieren und den IRQ anschließend aktivieren. Falls es sich um eine Multiprozessorumgebung handelt, so wird sichergestellt, dass diese Methode auf dem Prozessor ausgeführt wird, für den dieses Schedulerobjekt registriert wurde.

interrupt

```
public void interrupt()
```

Um den Timer-Interrupt zu empfangen wird das Interface `jx.zero.FirstLevelIrqHandler` implementiert. Der Lowlevel-Scheduler muss deshalb die Methode `interrupt` bereitstellen, welche von einem Timer-Interrupt aktiviert wird. In dieser Methode wird üblicherweise eine Domain aktiviert (durch die Methode `activateDomain` des Portals `LLSchedulerSupport`).

Dazu sollte zunächst geprüft werden, ob die aktuelle Domain noch rechnen darf. Falls nicht, sollte eine andere Domain ausgewählt werden.

activate_currDomain

```
public void activate_currDomain()
```

Durch diese Funktion soll die aktuelle Domain aktiviert werden. Der Scheduler sollte jedoch prüfen, ob die Domain noch genügend Rechenzeit zur Verfügung hat. Die Methode wird beispielsweise aufgerufen nachdem ein Thread beendet wurde.

activate_nextDomain

```
public void activate_nextDomain()
```

In dieser Methode könnte zum Beispiel die Schedulingstrategie hinterlegt sein, denn am Ende dieser Methode sollte eine andere Domain aktiviert werden (mit Hilfe der Methode `activateDomain` des Portals `LLSchedulerSupport`), als bisher aktiv war. Diese Funktion wird z. B. von der Methode `yield` des Portals `HLSchedulerSupport` aufgerufen.

registerDomain

```
public void registerDomain(Domain domain)
```

Durch diese Methode wird dem Lowlevel-Scheduler eine Domain gemeldet, nachdem sie einen Highlevel-Scheduler für die entsprechende CPU registriert hat. Üblicherweise sollte die Domain in eine Warteschlange aufgenommen werden.

unregisterDomain

```
public void unregisterDomain(Domain domain)
```

Mit dieser Methode soll eine Domain wieder aus der Warteschlange des Lowlevel-Schedulers entfernt werden. Sie wird vom Kern aufgerufen, wenn eine Domain ihr Highlevel-Schedulerobjekt auswechselt. Nachdem eine Domain durch diese Methode abgemeldet wurde, darf sie nicht mehr aktiviert werden, bis sie wieder durch die Methode `registerDomain` angemeldet wird.

setTimeSlice

```
public void setTimeSlice(Domain domain, int time)
```

Mit dieser Methode wird eine Zeitscheibe für eine Domain festgelegt. Das heißt, dass die Domain `domain` von nun an immer in `time` Zeiteinheiten unterbrochen werden möchte. Die Länge einer Zeiteinheit kann durch die Methode `getTimeBaseInMicros` des Portals `HLSchedulerSupport` abgefragt werden. Da jedoch sämtliche Zeitintervalle in dieser Einheit angegeben werden – im speziellen auch bei `tuneTimer` – ist die genaue Größe meist irrelevant. Der Scheduler sollte das Zeitintervall zu den Daten der Domain vermerken und bei der Aktivierung dieser Domain den Timer entsprechend einstellen.

dump

```
public void dump()
```

Diese Methode dient nur zu Debug-Zwecken. Sie sollte alle nötigen Informationen ausgeben um mögliche Fehler auffinden zu können. Zur Unterstützung bietet das Portal `LLSchedulerSupport` die Methode `dumpHLSched` an.

A.1.2 Das Portal `LLSchedulerSupport`

Wie in Abschnitt 5.5.7 erwähnt, kann der Lowlevel-Scheduler nicht direkt mit einem Highlevel-Scheduler kommunizieren. Deshalb stellt der Kern (Domain Zero) ein Portal zur Verfügung, mit dessen Hilfe der Lowlevel-Scheduler verschiedene Interaktionen durchführen kann. Darunter befinden sich auch Methoden, um eine Domain zu aktivieren und um die Timer-Steuerung vorzunehmen.

preemptCurrentThread

```
CPUState preemptCurrentThread();
```

Ein Aufruf dieser Methode ist nur nach einem Timer-Interrupt sinnvoll. Der Lowlevel-Scheduler kann damit dem Highlevel-Scheduler des unterbrochenen Threads mitteilen, dass er verdrängt wurde; die Methode `preempted` des Highlevel-Schedulers wird ausgeführt. Im Anschluss an einen Aufruf dieser Methode sollte der Lowlevel-Scheduler eine andere Domain aktivieren.
Die Methode liefert als Rückgabe den unterbrochenen Thread (als ein Objekt vom Typ `CPUState`).

interruptCurrentThread

```
CPUState interruptCurrentThread();
```

Auch diese Methode sollte nur nach einem Timer-Interrupt aufgerufen werden. Sie führt dazu, dass die Methode `interrupted` des Highlevel-Schedulers ausgeführt wird, der für den aktuell unterbrochenen Thread zuständig ist. Der Lowlevel-Scheduler sollte diese Methode nur aufrufen, wenn er der zuletzt aktivierten Domain erneut Rechenzeit zuteilt. Durch diese Methode soll das Ende einer domainspezifischen Zeitscheibe mitgeteilt werden.

activateDomain

```
void activateDomain(Domain active);
```

Wenn der Lowlevel-Scheduler eine Domain ausgewählt hat, so kann er sie mit Hilfe dieser Methode aktivieren. Da diese Methode nicht mehr zurückkehrt, stellt sie auch die Beendigung des Timer-Interrupthandlers dar.

activateIdleThread

```
void activateIdleThread();
```

Wenn bei einem neuen Lowlevel-Scheduler noch keine Domains registriert sind, so kann er durch diese Methode den Idle-Thread aktivieren. Dies ist nötig, damit die Interrupts wieder freigeschaltet werden.

getDomainTimeslice

```
int getDomainTimeslice();
```

Aus historischen Gründen kann bei der Konfiguration des Kernels die Länge einer Zeitscheibe festgelegt werden. Durch diese Methode kann der Wert abgefragt werden. Ein Lowlevel-Scheduler kann den so ermittelten Wert als Zeitscheibenlänge einer Domain verwenden.

tuneTimer

```
void tuneTimer (int time);
```

Durch diese Methode wird die Zeit bis zum nächsten Timer-Interrupt festgelegt. Die Einheit kann durch die Methode `getTimeBaseInMicros` des Portals `HLSchedulerSupport` abgefragt werden.

readTimer

```
int readTimer();
```

Mit Hilfe dieser Methode kann man die Zeit abfragen, die seit dem letzten Timer-Interrupt vergangen ist. Hier wird dieselbe Zeiteinheit zugrundegelegt, wie bei allen anderen Methoden die in diesem Anhang beschrieben werden.

dumpHLSched

```
void dumpHLSched (Domain domain);
```

Wenn der Lowlevel-Scheduler Informationen über den Highlevel-Scheduler einer Domain ausgeben will, so kann er diese Methode aufrufen. Daraufhin wird die Methode `dump` des Highlevel-Schedulers ausgeführt, der für die entsprechende Domain beim Kern registriert ist. Üblicherweise wird `dumpHLSched` in der Methode `dump` des Lowlevel-Schedulers für alle registrierten Domains aufgerufen.

A.2 Highlevel-Scheduler

Ein Highlevel-Scheduler wird durch das Interface `jx.zero.scheduler.HighLevelScheduler` spezifiziert. Die Methoden, die ein Highlevel-Scheduler zur Verfügung stellen muss bzw. kann, wurden bereits in Tabelle 5.1 aufgezählt. Der Vollständigkeit halber werden sie hier nochmals beschrieben.

A.2.1 Das Basisinterface `HighLevelScheduler`

created

```
public void created (CPUState thread);
```

Diese Methode wird aufgerufen, wenn ein neuer Thread erzeugt wird. Sie ist die einzige Informationsmethode, die das Basisinterface vorschreibt. Während dieser Methode sind die Interrupts deaktiviert. Die Laufzeit der Methode muss deshalb sehr kurz sein. Üblicherweise hängt der Highlevel-Scheduler den Thread – den er als Parameter erhält – in eine Warteschlange ein.

activated

```
public void activated();
```

Wenn der Lowlevel-Scheduler eine Domain aktiviert, so wird diese Methode des entsprechenden Highlevel-Schedulers ausgeführt. Sie läuft in einem speziellen Schedulingthread bei aktivierten Interrupts ab. Beendet werden sollte die Methode, indem ein Thread der Domain aktiviert wird. Dazu stellt das Interface `HLSchedulerSupport` die Methode `activateThread` zur Verfügung.

Scheduler_preempted

```
public boolean Scheduler_preempted();
```

Wenn die Zeitscheibe der Domain abgelaufen ist, wird der Lowlevel-Scheduler die Methode `preemptCurrentThread` aufrufen. Dort wird geprüft, ob der aktuell unterbrochene Aktivitätsträger der Schedulingthread des Highlevel-Schedulers ist. Falls dies zutrifft, wird, anstatt der Methode `preempted`, die Methode `Scheduler_preempted` aufgerufen.

Der Highlevel-Scheduler weiß dadurch, dass die Methode `activated` unterbrochen wurde. Es existieren nun zwei Situationen:

1. Die Methode `activated` wurde so entwickelt, dass sie an beliebiger Stelle abgebrochen werden kann, ohne dass die Datenstrukturen des Schedulers inkonsistent werden. Oder durch die Methode `activated` werden genügend Informationen gespeichert, um eventuell veränderte Daten wieder in einen konsistenten Zustand zu bringen.
   In diesem Fall kann der Scheduler durch den Rückgabewert `true` dem System mitteilen, dass er die Situation handhaben kann.

2. Ein Abbruch der Methode `activated` würde inkonsistente Daten erzeugen, wodurch der Scheduler nicht mehr lauffähig wäre.
   In dieser Situation sollte der Scheduler `false` zurückliefern. Das System wird dann bei der nächsten Aktivierung der Domain den Schedulingthread an der unterbrochenen Stelle fortfahren.

Scheduler_interrupted

```
public boolean Scheduler_interrupted();
```

Äquivalent zu `Scheduler_preempted` mit dem Unterschied, dass der Lowlevel-Scheduler die Methode `interruptCurrentThread` aufgerufen hat. Die Domain kann also damit rechnen unmittelbar danach wieder aktiviert zu werden. Der Rückgabewert hat die gleiche Bedeutung wie bei `Scheduler_preempted`.

registered

```
public void registered();
```

Diese Funktion wird aufgerufen, nachdem der Highlevel-Scheduler beim Lowlevel-Scheduler registriert wurde. Falls es sich um eine Multiprozessorumgebung handelt, so wird sichergestellt, dass diese Methode auf dem Prozessor ausgeführt wird, für den dieses Schedulerobjekt registriert wurde.

dump

```
public void dump();
```

Dient nur zu Test-Zwecken. Kann über die Methode `dumpHLSched` des Portals `LLSchedulerSupport` aufgerufen werden.

A.2.2 Optionale Interfaces für Highlevel-Scheduler

Im Folgenden werden alle Interfaces vorgestellt, die von einem Highlevel-Scheduler optional implementiert werden können (siehe Abschnitt 5.5.6). In allen ist lediglich eine Methode definiert. Bei fast allen handelt es sich um Informationsmethoden, die einem Highlevel-Scheduler den Zustandswechsel eines seiner Thread mitteilen können (siehe auch Tabelle 5.1). Ausnahme hiervon ist die Methode `switchTo`.

Bis auf `switchTo` werden alle Methoden unter Interruptausschluss ausgeführt. Der Code innerhalb dieser Methoden muss dementsprechend kurz sein.

interface HLS_preempted

```
public void preempted (CPUState thread);
```

Diese Methode wird aufgerufen, wenn der Lowlevel-Scheduler die Methode `preemptCurrentThread` des Portals `LLSchedulerSupport` aufruft. Normalerweise geschieht das nach einem Timer-Interrupt, wenn die Zeitscheibe der Domain abgelaufen ist. Als Parameter wird der Thread mitgegeben, der durch den Timer unterbrochen wurde.

interface HLS_interrupted

```
public void interrupted(CPUState thread);
```

Äquivalent zu `preempted`, jedoch hat der Lowlevel-Scheduler die Methode `interruptCurrentThread` verwendet, um dem Highlevel-Scheduler die Unterbrechung des Threads mitzuteilen. Üblicherweise wird der Lowlevel-Scheduler sofort anschließend wieder diese Domain aktivieren.

interface HLS_blocked

```
public void blocked (CPUState thread);
```

Wird ein Thread blockiert (z. B. durch den Aufruf von `CPUManager::block()`), so wird dieses Ereignis durch die Methode `blocked` dem entsprechenden Highlevel-Scheduler gemeldet. Als Argument erhält der Scheduler dabei den blockierten Thread.

interface HLS_unblocked

```
public void unblocked (CPUState thread);
```

Wenn ein blockierter Thread wieder deblockiert wird, so wird diese Methode des entsprechenden Schedulers aufgerufen.

HLS_destroyed

```
public void destroyed (CPUState thread);
```

Diese Methode informiert den Scheduler, dass einer seiner Threads beendet wurde. Der entsprechende Thread wird als Parameter übergeben.

interface HLS_yielded

```
public void yielded (CPUState thread);
```

Gibt ein Thread seine Rechenzeit freiwillig ab, (z. B. durch einen Aufruf von `Thread::yield()`), so wird diese Methode "seines" Schedulers aufgerufen.

interface HLS_blockedInPortalCall

```
public void blockedInPortalCall (CPUState thread);
```

Durch diese Methode wird dem Scheduler mitgeteilt, dass ein Thread einen Portalaufruf gestartet hat, der jedoch nicht sofort ausgeführt werden kann (siehe Abschnitt 6.5.1). Da der Thread blockiert ist, bis ein Ergebnis des Portalaufrufes vorliegt, ist diese Methode ein Spezialfall von `blocked`.

interface HLS_portalCalled

```
public boolean portalCalled(CPUState thread);
```

Diese Methode wird anstatt `blockedInPortalCall` aufgerufen, falls ein Thread einen Portalaufruf startet und dieser sofort durchgeführt werden könnte. Der Highlevel-Scheduler kann durch den Rückgabewert dem System mitteilen, ob er Rechenzeit an die Domain des Portals abgeben möchte. Dabei bedeutet `false`, dass keine Zeit für die Ausführung des Portals zur Verfügung gestellt werden soll; bei einer Rückgabe von `true` wird die verbleibende Zeitscheibe der Domain des Portals überlassen (siehe Methode `switchTo`). In jedem Fall ist der Thread, der durch den Parameter spezifiziert ist, blockiert.

interface HLS_switchTo

```
public void switchTo(CPUState thread);
```

Wird einer Domain Zeit von einer anderen Methode zur Verfügung gestellt (z. B. im Rahmen eines Portalaufrufes), so ist damit immer der Wunsch verbunden einen bestimmten Thread auszuführen. Diese Methode dient dazu, dem Highlevel-Scheduler eine solche Situation mitzuteilen. Als Parameter wird der Thread übergeben, der ausgeführt werden soll. Der Scheduler kann jedoch entscheiden welchen Thread er aktiviert und behält somit die volle Kontrolle über die Aktivitätsträger einer Domain.

Während dieser Funktion sind die Interrupts nicht deaktiviert, sie wird vom Schedulingthread des entsprechenden Highlevel-Schedulers ausgeführt. Die Methode muss entweder durch einen Aufruf von `activateThread` oder `yield` des Portals `HLSchedulerSupport` beendet werden.

interface HLScheduler_runnables

Dieses Interface stellt eine vordefinierte Konfiguration dar. Es vereint die Interfaces `HighLevelScheduler`, `HLS_preempted`, `HLS_interrupted`, `HLS_yielded` und `HLS_unblocked`.

Es ist als Beispiel gedacht, um einen einfachen Round-Robin-Scheduler – wie in Abschnitt 5.5.6 beschrieben – zu implementieren.

A.2.3 Das Portal `HLSchedulerSupport`

Das Portal `HLSchedulerSupport` umfasst Methoden, die speziell für die Unterstützung eines Highlevel-Schedulers entwickelt wurden. Darunter befindet sich beispielsweise die Möglichkeit einen Thread zu aktivieren, oder eine domainspezifische Zeitscheibe festzulegen.

activateThread

```
void activateThread(CPUState thread);
```

Mit Hilfe dieser Methode kann ein Highlevel-Scheduler die Rechenzeit an einen anderen Thread abgeben. Dazu wird der angegebene Aktivitätsträger `thread` aktiviert. Zu beachten ist, dass der aktuelle Thread nicht gesichert wird, da diese Methode nur im Schedulingthread verwendet werden sollte (d. h. in den Methoden `activated` oder `switchTo` des Highlevel-Schedulers).

yield

```
void yield();
```

Falls der Highlevel-Scheduler keinen lauffähigen Thread hat, so kann er durch diese Methode dem Lowlevel-Scheduler mitteilen, dass die nächste Domain aktiviert werden kann. Dies ist eine alternative Beendigung der Methode `activated` des Highlevel-Schedulers.

getTimeBaseInMicros

```
int getTimeBaseInMicros();
```

Liefert die Zeiteinheit in Mikrosekunden zurück. Alle Zeitangaben beziehen sich auf diese Zeiteinheit.

getDomainTimeslice

```
int getDomainTimeslice();
```

Gibt die Länge einer Zeitscheibe in Zeiteinheiten zurück, die eine Domain den Prozessor belegen darf.

setMyTimeslice

```
void setMyTimeslice (int time);
```

Durch diese Methode kann ein Highlevel-Scheduler ein Zeitintervall definieren, in das er seine verfügbare Zeit (siehe `getDomainTimeslice`) einteilen möchte. Eigentlich führt diese Methode lediglich zu einem Aufruf der Methode `setTimeSlice` des Lowlevel-Schedulers. Dieser wird dann, je nach Implementierung, die Domain innerhalb ihrer Zeitscheibe entsprechend oft unterbrechen, um dem Highlevel-Scheduler die Möglichkeit zu geben einen anderen Thread zu aktivieren.

storeThreadInfo, getThreadInfo

```
void storeThreadInfo(CPUState thread, Object data);
Object getThreadInfo (CPUState thread);
```

Mit Hilfe dieser Methoden hat ein Highlevel-Scheduler die Möglichkeit ein beliebiges Objekt mit einem Thread zu verknüpfen und dieses wieder abzufragen. Dadurch können Informationen direkt beim entsprechenden Thread gespeichert werden.

getThreadStatus

```
int getThreadStatus (CPUState thread);
```

Durch diese Methode kann der Scheduler den Status eines Aktivitätsträger (z. B. blockiert, lauffähig, ...) abfragen. Wird ein Highlevel-Scheduler über alle Ereignisse informiert, so wird er diese Methode nicht benötigen.

Die möglichen Rückgabewerte sind in `jx.zero.HLSchedulerSupport` definiert.

dumpThread

```
void dumpThread (CPUState thread);
```

Diese Methode gibt Informationen über einen Thread aus.

A.3 Das Portal `SMPCPUManager`

Um die Fähigkeiten eines Multiprozessorsystems zu nutzen, stellt die Domain Zero das Portal `SMPCPUManager` zur Verfügung. Im Folgenden sollen die wichtigsten Methoden betrachtet werden:

getNumCPUs

```
int getNumCPUs();
```

Diese Methode gibt die Anzahl der Prozessoren zurück, die eine Domain verwenden darf.

getCPU, getMyCPU

```
CPU getCPU(int i);
CPU getMyCPU();
```

Diese Methoden liefern ein CPU Objekt zurück. Mit `getCPU` kann man das Objekt eines bestimmten Prozessors erhalten, durch `getMyCPU` wird das Objekt der aktuellen CPU in Erfahrung gebracht.

unblock

```
boolean unblock(CPUState thread, CPU cpu);
```

Durch diese Methode ist es dem Benutzer möglich einen Thread auf einem bestimmten Prozessor zu deblockieren. Das heißt, man kann steuern, welcher Highlevel-Scheduler die Nachricht `unblocked` erhalten soll.

swap_HLScheduler

```
swap_HLScheduler(CPU cpu, Domain domain,
                 HighLevelScheduler oldSched,
                 HighLevelScheduler newSched);
```

Mit Hilfe dieser Methode kann ein Highlevel-Schedulerobjekt zur Laufzeit ausgetauscht werden. Als Parameter wird ein CPU-Objekt erwartet, die Domain deren Schedulerobjekt getauscht werden soll, der alte Scheduler und ein neues Schedulerobjekt. Die Funktion trägt das neue Objekt in den Domainkontrollblock ein, das alte Schedulerobjekt wird dabei lediglich zur Authentifikation benötigt. Für die Übertragung der Schedulingdaten vom alten Scheduler in den neuen, ist der Aufrufer selbst zuständig.

Index

82093AA I/O-APIC 12
82489DX local APIC 12

A
Adressierung 17
    logische 18
    physikalische 17
Aktivierungsmethoden 38
Aktivierungssequenz 19
APIC 11
    82093AA 12
    82489DX 12
    I/O 11
    Konfiguration 23
    lokaler 11
APIC-Bus 18
Arbitrierung 18

B
Benachrichtigungen 37
    konfigurieren 45
Bootloader 21
Bytecode-Verifier 40, 55

C
cachekohärent 7
Cachekohärenz 7, 10
Cluster Modell 18
    flaches 18
    hierarchisches 18
Co-Scheduling 34
Compiler-Plug-In 52

D
Domain 3
Domain Zero 4, 34, 48, 71
Domainwechsel 37, 48, 49

E
Ereignislisten 49

F
flaches Modell 18
Floating Pointer Structure 14

G
Gang-Scheduling 34

H
heterogene Systeme 6
Highlevel-Scheduler 32
homogene Systeme 6

I
I/O-APIC 11
ICC-Bus 10
Identifikation 17
Idle-Thread 21
IMCR 13, 15, 19
in-service-Register 24
Informationsmethoden 37
INIT IPI 20
inlined Portal 4, 48
Interprozesskommunikation 56
Interprozessorinterrupt 19, 25
Interrupt
    Controller
        APIC 11
        PIC 11
    Interprozessor- 19, 25
    Self- 25
    Vektornummer 23
Interrupt-Request-Register 24
Interruptsystem 5, 23
Intervalltimer 51
IPC 56

IPI 19, 20, 25

K

Konfigurationstabelle 14

kooperatives Scheduling 58

L

Lastausgleich 34

local Timer 26

logische Adressierung 18

lokaler APIC 11

Lowlevel-Scheduler 32

M

MESI-Protokoll 10

message passing 7

Methodenaufruf

  domainübergreifend 56

Multiprozessor-Systeme 6

N

no remote memory access (NORMA) 7

no uniform memory access (NUMA) 6

NORMA 7

NUMA 6

P

physikalische Adressierung 17

PIC 11

PIIX4 11

Portal 4, 56

  inlined 4, 48

Protected-Mode 20

R

Real-Mode 20

Real-Time-Clock 51

Ressourcenbeschränkung 61

S

Scheduler

  globaler 30

  Highlevel 32

  Lowlevel 32

Schedulerportal 43

Scheduling

  Co- 34

  Gang- 34

  kooperatives 58

Self-Interrupt 25

shared memory 7

SMP 10

Startcode 20

STARTUP IPI 20

Stellvertreter 49, 56

Stub 49

T

Timer

  local 26

  programmierbarer Intervall- 51

U

UMA 6

uniform memory access (UMA) 6

V

Vektornummer 23

Z

Zeigerstruktur 14

Zeitscheibe 50

  individuelle 52

Zeitscheibenstrategie 52

Literaturverzeichnis

[Alt01] ALT, MARTIN: *Ein Bytecode-Verifier zur verifikation von Betriebssystemkomponenten.* Diplomarbeit, Universität Erlangen, IMMD IV, Juli 2001.

[FB96] FORD, BRYAN und ERICH STEFAN BOLEYN: *Multiboot Standard*, Version 0.6, März 1996.
http://www.uruk.org/orig-grub/boot-proposal.html.

[FS96] FORD, BRYAN und SAI SUSARLA: *CPU Inheritance Scheduling.* In: *Proceedings of the 2^{nd} Symposium on Operating Systems Design and Implementation*, 1996.

[GK00] GOLM, MICHAEL und JÜRGEN KLEINÖDER: *JX: Eine adaptierbare Java-Betriebssystemarchitektur.* GI-Herbsttagung: Mobile Computing, München, November 2000.

[GKB01] GOLM, MICHAEL, JÜRGEN KLEINÖDER und FRANK BELLOSA: *Beyond Address Spaces - Flexibility, Performance, Protection, and Resource Management in the Type-Safe JX Operating System.* Hot Topics in Operating Systems VIII, Elmau, Mai 2001.

[Han93] HANDY, JIM: *The Cache Memory Book.* Academic Press, London, 1993.

[Hof89] HOFMANN, WILHELM: *Erweiterung des UNIX-Betriebssystems für Multiprozessoren.* In: *Arbeitsberichte des Instituts für mathematische Maschinen und Datenverarbeitung*, Band 22, Nummer 10, Seiten 1–61. Friedrich Alexander Universität Erlangen-Nürnberg, Erlangen, März 1989.

[Hof90] HOFMANN, WILHELM: *Die Koordinierung in Betriebssystemen für Multiprozessoren.* In: *Arbeitsberichte des Instituts für mathematische Maschinen und Datenverarbeitung*, Band 23, Nummer 10. Friedrich Alexander Universität Erlangen-Nürnberg, Erlangen, Juli 1990.

[Int96] INTEL CORPORATION: *82093AA I/O Advanced Programmable Interrupt Controller (IOAPIC)*, Mai 1996. Order Number: 290566-001.

[Int97a] INTEL CORPORATION: *82371AB PCI-TO-ISA / IDE Xcelerator (PIIX4)*, April 1997. Order Number: 290562-001.

[Int97b] INTEL CORPORATION: *MultiProcessor Specification, Version 1.4*, Mai 1997. Order Number: 242016-006.

[Int98] INTEL CORPORATION: *P6 Family of Processors, Hardware Developer's Manual*, September 1998. Order Number: 244001-001.

[Int99a] Intel Corporation: *Intel Architecture Software Developer's Manual, Volume 2: Instruction Set Reference*, 1999. Order Number: 243191-002.

[Int99b] Intel Corporation: *Intel Architecture Software Developer's Manual, Volume 3: System Programming*, 1999. Order Number: 243192-002.

[Int00] Intel Corporation: *Intel Processor Identification and the CPUID Instruction*, Mai 2000. Order Number: 241618-015.

[KR94] Kleinöder, Jürgen und Thomas Riechmann: *Hierarchical Schedulers in the PM System-Architecture*. Technischer Bericht TR-14-94-16, Universität Erlangen, IMMD IV, Juni 1994.

[MC91] Mellor-Crummey, John M.: *Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors*. ACM Transactions on Computer Systems, 9(1):21–65, Februar 1991.

[Mes94] Messmer, Hans-Peter: *Pentium*. Addison-Wesley, Bonn, 1 Auflage, 1994.

[Mes98] Messmer, Hans-Peter: *PC Hardware*. Addison-Wesley, Bonn, 5. Auflage, 1998.

[MS96] Michael, Maged M. und Michael L. Scott: *Simple, Fast and Practical Non-Blocking and Blocking Concurrent Queue Algorithms*. In: *Proceedings of the 15th Annual Principles of Distributed Computing*, Philadelphia, Mai 1996.

[Ous82] Ousterhout, John K.: *Scheduling Techniques for Concurrent Systems*. In: *Proceedings of the 3rd International Conference on Distributed Computer Systems*, Seiten 22–30, Miami, Oktober 1982. IEEE Computer Society Press.

[Ras87] Rashid, R. F.: *Design for Parallel Architectures*. UNIX Review, Seiten 36–43, April 1987.

[Rie94] Riechmann, Thomas: *Hierarchische Scheduler in verteilten, objektorientierten Systemen*. Diplomarbeit, Universität Erlangen, IMMD IV, Juli 1994.

[RK96] Riechmann, Thomas und Jürgen Kleinöder: *User-Level Scheduling with Kernel Threads*. Technischer Bericht TR-14-96-05, Universität Erlangen, IMMD IV, Juni 1996.

[TZ00] Tsigas, Philippas und Yi Zhang: *A Simple, Fast and Scalable Non-Blocking Concurrent FIFO queue for Shared Memory Multiprocessor Systems*. Technischer Bericht TR-2000-01, Chalmers University of Technology, 2000.

[Val95] Valois, John D.: *Lock-Free Linked Lists Using Compare-and-Swap*. In: *Proceedings of the 14th Annual Principles of Distributed Computing*, Seiten 214–222. ACM, Inc., August 1995.

[Waw01] Wawersich, Christian: *Design und Implementierung eines Profilers und optimierenden Compilers für das Betriebssystem JX*. Diplomarbeit, Universität Erlangen, IMMD IV, April 2001.

[WM95] Wohak, Bertram und Reinhold Maurus: *80x86/Pentium Assembler*. IWT Verlag, Bonn, 1 Auflage, 1995.