MPStuBS
Eine multiprozessorfähige Variante des OOStuBS-Lehrbetriebssystems

Studienarbeit

von

Andreas Schweikart

geboren am 03.08.1983 in Nürnberg

Lehrstuhl für Informatik 4
Friedrich-Alexander Universität Erlangen-Nürnberg

Betreuer:	Dipl.-Inf. Daniel Lohmann Dipl.-Inf. Wanja Hofer
Beginn der Arbeit:	01.01.2008
Ende der Arbeit:	31.03.2008

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 _____ , _____

Zusammenfassung

Multi-Prozessor-Systeme sind bereits weit verbreitet und bieten heutzutage die wahrscheinlich einfachste Möglichkeit, die Leistung von Computersystemen weiter zu steigern. Deshalb müssen aktuelle Betriebssysteme die Komponenten eines Multi-Prozessor-Systems erkennen und gegebenenfalls verwenden.

In der vorliegenden Studienarbeit wird das bereits existierende Lehrbetriebssystem OOStuBS (Objektorientiertes Studentenbetriebssystem) zu MPStuBS (Multi-Prozessor-Studentenbetriebssystem) erweitert. Im Gegensatz zu OOStuBS muss MPStuBS die Multi-Prozessor-Architektur eines Systems erkennen und die dort zusätzlich vorhandenen Komponenten konfigurieren. Zusätzlich benötigt MPStuBS eine Erweiterung der OOStuBS-Synchronisationsmechanismen um *Spinlocks*. Die Rechenlast verteilt MPStuBS gleichmäßig auf alle vorhandenen Prozessoren.

MPStuBS wurde auf dem Emulator Bochs und mehreren verschiedenen Hardware-Plattformen erfolgreich getestet.

Inhaltsverzeichnis

Kapitel 1

Einführung in Multi-Processing

Ein Multi-Prozessor-System besteht aus mindestens zwei Prozessoren, die gemeinsam ein Computersystem bilden. Das Ziel eines solchen Systems ist es, die Gesamtleistung zu steigern, indem Aufgaben parallel ausgeführt werden. Vergleicht man ein Multi-Prozessor-System und ein Mono-Prozessor-System, die beide den gleichen Prozessortyp verwenden, führt das Multi-Prozessor-System eine Aufgabe nicht unbedingt viel schneller aus als das Mono-Prozessor-System, es ist jedoch in der Lage, parallel dazu andere Aufgaben zu bearbeiten. Im Gegenteil zu Mono-Prozessor-Systemen, bei denen man versucht, eine Leistungssteigerung durch erhöhte Rechengeschwindigkeit zu erreichen, besteht die Leistungssteigerung bei einem Multi-Prozessor-System darin, dass pro Zeiteinheit mehrere Aufgaben bearbeitet werden.

Dass der Markt bis einschließlich 2005 von Mono-Prozessor-Systemen dominiert wurde, liegt vor allem daran, dass das Potential von Multi-Prozessor-Systemen schwer zu erkennen war. Da die existierende Software für Mono- Prozessor-Systeme ausgelegt war, war der Leistungszuwachs bei Multi-Prozessor-Systemen verhältnismäßig gering. Um ihn zu maximieren, hätte die bestehende Software für Multi-Prozessor-Systeme portiert werden müssen, was viel komplizierter ist, als sie für schnellere Mono-Prozessoren zu portieren. Außerdem hat sich die Leistungsfähigkeit von Mono-Prozessor-Systemen so schnell gesteigert, dass die Notwendigkeit Multi-Prozessor-Systeme zu verwenden lange nicht gegeben war.

Heutzutage ist man soweit, dass es nahezu unmöglich und vor allem sehr teuer ist, schnellere Prozessoren zu entwickeln. Deshalb bieten Multi-Prozessor-Systeme eine ernst zu nehmende Alternative, da sie verhältnissmäßig billig sind, und ihre Leistungsfähigkeit, durch die Möglichkeit weitere Prozessoren hinzuzufügen, nahezu uneingeschränkt ist. Dual-Core-Prozessoren sind bereits weit verbreitet, und bekannte Betriebssysteme wie Unix, Windows oder Mach sind bereits multiprozessorfähig. Darüberhinaus ist moderne Software für Multi-Prozessor-Systeme ausgelegt, wodurch das gewünschte Ziel, doppelte Leistung bei doppelter Anzahl an Prozessoren, nahezu erreicht wird.

Aufgrund der also steigenden Bedeutung von Multi-Prozessor-Systemen sollte in dieser Studienarbeit das schon existierende Mono-Prozessor-Betriebssystem OO-

StuBS (Objektorientiertes Studenten-Betriebssystem) zu MPStuBS (Multi-Prozessor-Studenten-Betriebssystem) erweitert werden. Dieses MPStuBS soll ein Multi-Prozessor-System gegebenenfalls erkennen und sowohl Benutzerprogramme, als auch Aufgaben des Betriebssystems auf alle Prozessoren verteilen.

Die vorliegende Arbeit ist folgendermaßen gegliedert: Zunächst werden einige Multi-Prozessor-Architekturen vorgestellt, und die für MPStuBS verwendete festgelegt. Danach werden in Kapitel 3 die nötigen Hintergründe über die Hardware, den Speicher und das BIOS in einem Multi-Prozessor-System vorgestellt. Das Kapitel 4 beschreibt dann grundlegenden Konzepte eines Betriebssystems, die in OOStuBS verwendet wurden, und die auch für MPStuBS übernommen werden sollen, bevor in Kapitel 5 die Anforderungen an MPStuBS genau definiert werden. Das letzte Kapitel erläutert die Implementierung von MPStuBS.

Kapitel 2

Multi-Prozessor-Hardware-Architekturen

Ein Rechner besteht aus den drei Teilsystemen Prozessor, Speicher und Ein-/Ausgabesystem (I/O-System). Die Rechnerarchitektur wird durch die Beziehungen dieser Systeme untereinander definiert. Multi-Prozessor-Systeme werden anhand von drei Kriterien in unterschiedliche Klassen aufgeteilt [9]. Das erste Kriterium basiert auf dem Befehlsfluss. Arbeiten alle Prozessoren den gleichen Befehlsfluss ab, handelt es sich um eine *Single Instruction Multiple Data* (SIMD)-Architektur, arbeiten sie jeweils einen eigenen Befehlsfluss ab, handelt es sich um eine *Multiple Instruction Multiple Data* (MIMD)-Architektur. Die Art und Weise der Kommunikation der Prozessoren untereinander klassifiziert Multi-Prozessor-Systeme zusätzlich in *Message Passing*- und *Shared Memory*-Architekturen. Als letztes unterscheidet man zwischen Systemen, deren Prozessoren auf Busebene oder über ein Netzwerk verbunden sind. Das führt zu der Klassifizierung in *Tightly Coupled*- und *Loosely Coupled*-Architekturen.

2.1 SIMD- und MIMD-Architekturen

SIMD-Architekturen arbeiten einen Befehlsfluss parallel auf mehreren Datenelementen ab, d.h. jeder Prozessor erhält von einer zentralen Kontrolleinheit den gleichen Befehl, und führt diesen, synchron mit allen anderen Prozessoren auf unterschiedlichen Daten aus. Um Unterschiede in der Programmausführung auf einzelnen Prozessoren zu erzwingen, müssen lokale Bedingungen abgefragt werden. Nur so kann man Prozessor-spezifischen Code auch nur von den richtigen Prozessoren ausführen lassen. Da alle Prozessoren im „Gleichschritt“ arbeiten, führt das dazu, dass nicht ausgewählte Prozessoren auf andere warten müssen, und somit der hohe Grad an Parallelität leidet.

In einer MIMD-Architektur verfügt jeder Prozessor über einen eigenen Programmspeicher und arbeitet seinen eigenen Befehlsfluss asynchron zu den anderen Prozessoren ab. Erst beim Austausch von Ergebnissen untereinander ist es nötig, die

Prozessoren zu synchronisieren. Abbildungen 2.1 und 2.2 zeigen eine SIMD- und eine MIMD-Architektur.

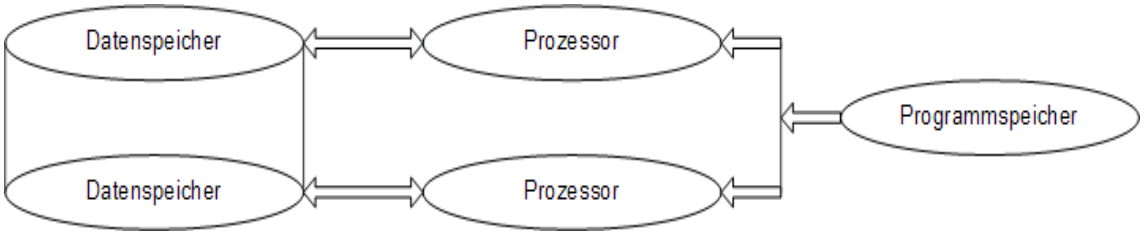


Abbildung 2.1: SIMD-Architektur (nach [9])

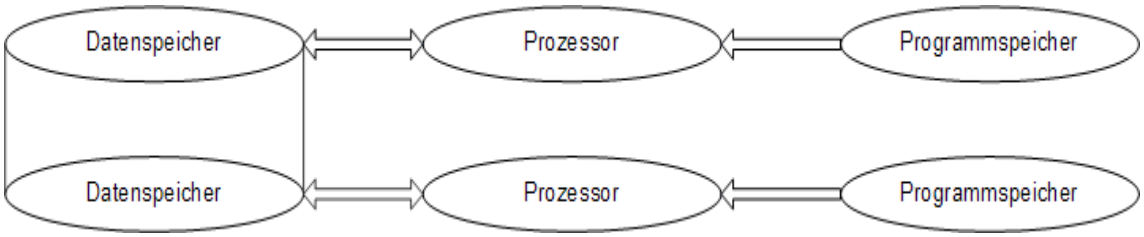


Abbildung 2.2: MIMD-Architektur (nach [9])

Die SIMD-Architektur hat einige Vorteile gegenüber der MIMD-Architektur, z.B. benötigt sie weniger zusätzliche Hardware-Komponenten, da keine Kommunikation der Prozessoren untereinander stattfindet. Des Weiteren ist keine Synchronisation der Prozessoren nötig, und sie arbeitet äußerst effektiv bei Aufgaben, die große Mengen von Daten gleichmäßig bearbeiten, ein passendes Beispiel dafür ist die Bildbearbeitung.

Auf der anderen Seite hat auch die MIMD-Architektur einige Vorteile. Die Auslastung der Prozessoren ist höher als bei SIMD-Architekturen, da kein Prozessor warten muss, während ein anderer eine Prozessor-spezifische Aufgabe ausführt. Darüberhinaus sind MIMD-Architekturen deutlich flexibler, da sie unabhängige Befehlsflüsse abarbeiten, weswegen sich der Trend stark in Richtung der MIMD-Architekturen bewegt.

2.2 *Message Passing*- und *Shared Memory*-Architekturen

Die Unterscheidung zwischen *Message Passing*- und *Shared Memory*-Architekturen beruht auf der Art und Weise, wie die Prozessoren eines Multi-Prozessor-Systems miteinander kommunizieren. In einem *Message Passing*-System arbeiten mehrere voneinander unabhängige Prozessoren zusammen, indem sie sich gegenseitig Nachrichten schicken. In einem *Shared Memory*-System sind die Prozessoren enger gekoppelt. Sie arbeiten auf dem gleichen Speicher, und die Kommunikation wird über gemeinsame Variablen oder gemeinsame Nachrichten-Puffer abgewickelt. Um einem

anderen Prozessor Daten zur Verfügung zu stellen, müssen sie lediglich in den gemeinsamen Speicher geschrieben werden. Im Gegensatz dazu müssen solche Daten in einem *Message Passing*-System in Nachrichten verpackt an den anderen Prozessor geschickt werden. Abbildungen 2.3 und 2.4 zeigen die Architekturen von *Message Passing*- und *Shared Memory*-Systemen.

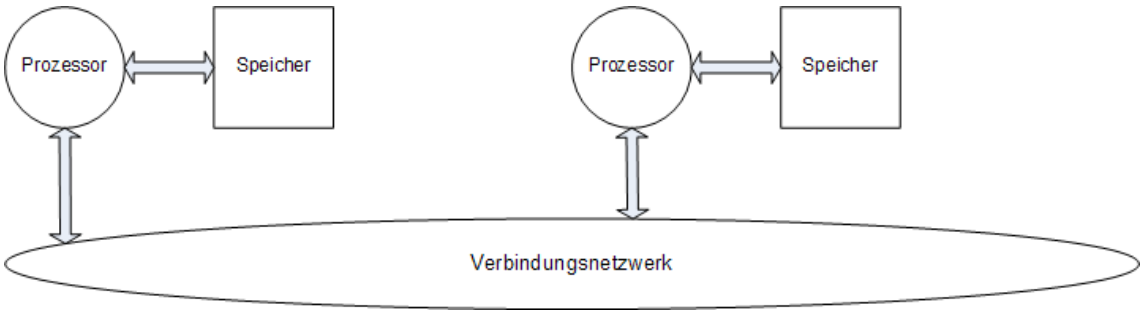


Abbildung 2.3: *Message Passing*-Architektur (nach [10])

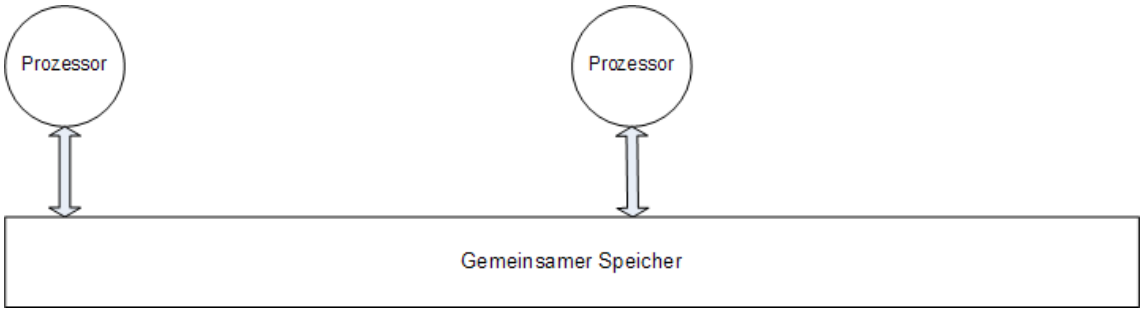


Abbildung 2.4: *Shared Memory*-Architektur (nach [10])

In *Message Passing*-Systemen ist die Kommunikation zwischen den Prozessoren direkt, der sendende Prozessor erstellt eine Nachricht und adressiert sie an den gewünschten Prozessor. Der empfangende Prozessor wird dann von der Ankunft einer Nachricht unterrichtet, und kann selbst entscheiden, was er damit tun möchte. In *Shared Memory*-Architekturen verläuft die Kommunikation indirekt. Der sendende Prozessor schreibt die Nachricht in den Speicher, und der empfangende Prozessor muss sie sich selber holen, was für den empfangenden Prozessor mit zusätzlichem Aufwand verbunden ist. Trotzdem ist der gesamte zusätzliche Aufwand für die Kommunikation bei *Shared Memory*-Architekturen geringer, da in *Message Passing*-Systemen das Versenden von Nachrichten über *System Calls* abläuft, die zusätzlichen Aufwand mit sich bringen.

Der größte Vorteil von *Message Passing*-Systemen ist, dass kaum zusätzliche Hardware benötigt wird. Schon vollkommen unabhängige PCs, die über die entsprechende Kommunikationssoftware verfügen, können als *Message Passing*-Multi-Prozessor-System gesehen werden.

Shared Memory-Architekturen haben den Vorteil, dass die Programmierung von Multi-Prozessor-Software verhältnismäßig einfach ist, und nur geringfügig von der Programmierung von Mono-Prozessor-Software abweicht.

2.3 *Tightly* und *Loosely Coupled*-Architekturen

Tightly Coupled-Architekturen sind Multi-Prozessor-Systeme, deren Prozessoren auf Busebene miteinander verbunden sind.

Loosely Coupled-Architekturen sind auch bekannt als Cluster, und basieren auf mehreren unabhängigen PCs, die über Hochgeschwindigkeits-Netzwerke (wie z.B. Gigabit Ethernet) miteinander verbunden sind.

2.4 *Tightly Coupled*, *Shared Memory*, *Symmetric* Multi-Prozessor

Die am häufigsten verwendete Multi-Prozessor-Architektur ist der *Tightly Coupled*, *Shared Memory*, *Symmetric* Multi-Prozessor, auch SMP genannt. Das im Laufe dieser Arbeit entwickelte Betriebssystem beruht auf einer solchen Architektur. Abbildung 2.5 zeigt einen SMP.

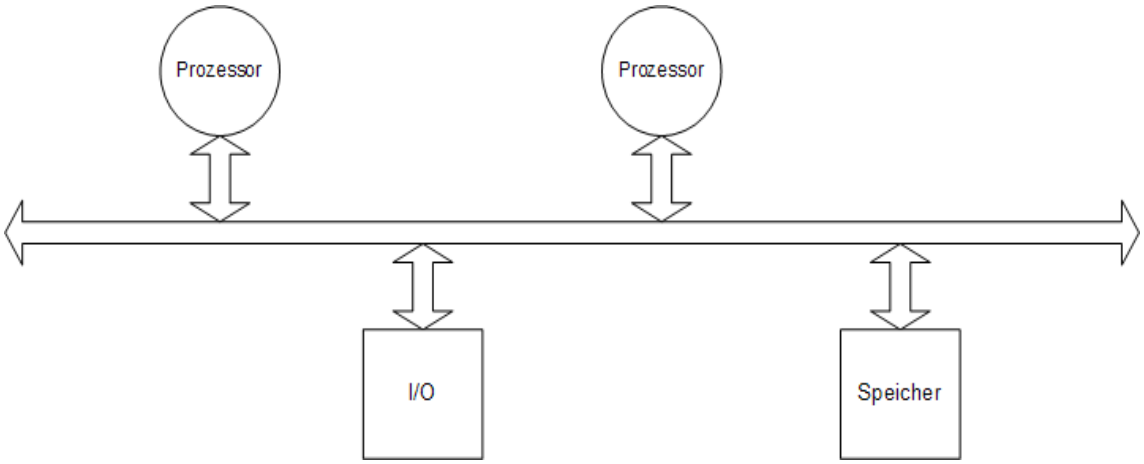


Abbildung 2.5: *Tightly Coupled*, *Shared Memory*, *Symmetric* Multi-Prozessor (nach [10])

Ein SMP ist ein MIMD-System. Alle Prozessoren, der Speicher und die I/O-Geräte sind eng gekoppelt, d.h. sie sind alle über einen Bus miteinander verbunden. Die Bandbreite dieses Busses limitiert die Anzahl der Prozessoren, da der Bus von allen Geräten in Anspruch genommen wird, und bei zu geringer Bandbreite paralleles Arbeiten nicht effizient möglich ist.

Der SMP verfügt über einen gemeinsamen Speicher (*Shared Memory*) auf den alle Geräte zugreifen können. Daten, die ein Prozessor in den Speicher schreibt, sind für alle anderen Prozessoren sofort zugänglich.

In einem symmetrischen Multi-Prozessor-System können alle Komponenten gleichberechtigt auf den gemeinsamen Speicher zugreifen. Bei gleichzeitigen Zugriffen wird

fair entschieden, wer an der Reihe ist, und es ist gewährleistet, dass Zugriffe sich nicht gegenseitig behindern.

2.5 Zusammenfassung

Multi-Prozessor-Architekturen werden in SIMD- und MIMD-, *Message Passing* und *Shared Memory* und *Tightly* und *Loosly Coupled*-Architekturen unterteilt. Der SMP ist die am häufigsten verwendete Architektur.

Im Folgenden wird genauer auf die einzelnen Komponenten eines Multi-Prozessor-Systems eingegangen.

Kapitel 3

Systemüberblick

Bevor das Betriebssystem entwickelt werden kann, müssen einige Hintergründe über das System bekannt sein. In diesem Kapitel wird auf die Funktionsweise der Multi-Prozessor-Komponenten eingegangen und beschrieben, in welchem Zustand sich das System befindet, wenn das BIOS die Kontrolle an das Betriebssystem übergibt.

3.1 Hardwareüberblick

In Kapitel 2 wurden einige Multi-Prozessor-Architekturen vorgestellt. MPStuBS basiert auf einem *Tightly Coupled*, *Shared Memory*, *Symmetric* Multi-Prozessor, d.h. alle Prozessoren sind gleichwertig und funktional identisch. Sie können mit jedem anderen Prozessor kommunizieren und es gibt keine Hierarchie unter den Prozessoren. Folgende Hardware-Komponenten werden im Gegensatz zu einem Mono-Prozessor-System verwendet:

- Mehrere Prozessoren
- Ein lokaler APIC (*Advanced Programmable Interrupt Controller*) pro Prozessor
- Mindestens ein I/O-APIC
- Gemeinsamer Speicher

Abbildung 3.1 zeigt die Architektur eines Multi-Prozessor-Systems. Im Folgenden wird die Funktionsweise der oben genannten Komponenten, wie in [1] beschrieben, vorgestellt.

3.1.1 Prozessoren

Obwohl alle Prozessoren gleichwertig sind, werden sie in einem Multi-Prozessor-System in zwei Gruppen unterteilt, den *Bootstrap*-Prozessor (BSP) und die *Appli-*

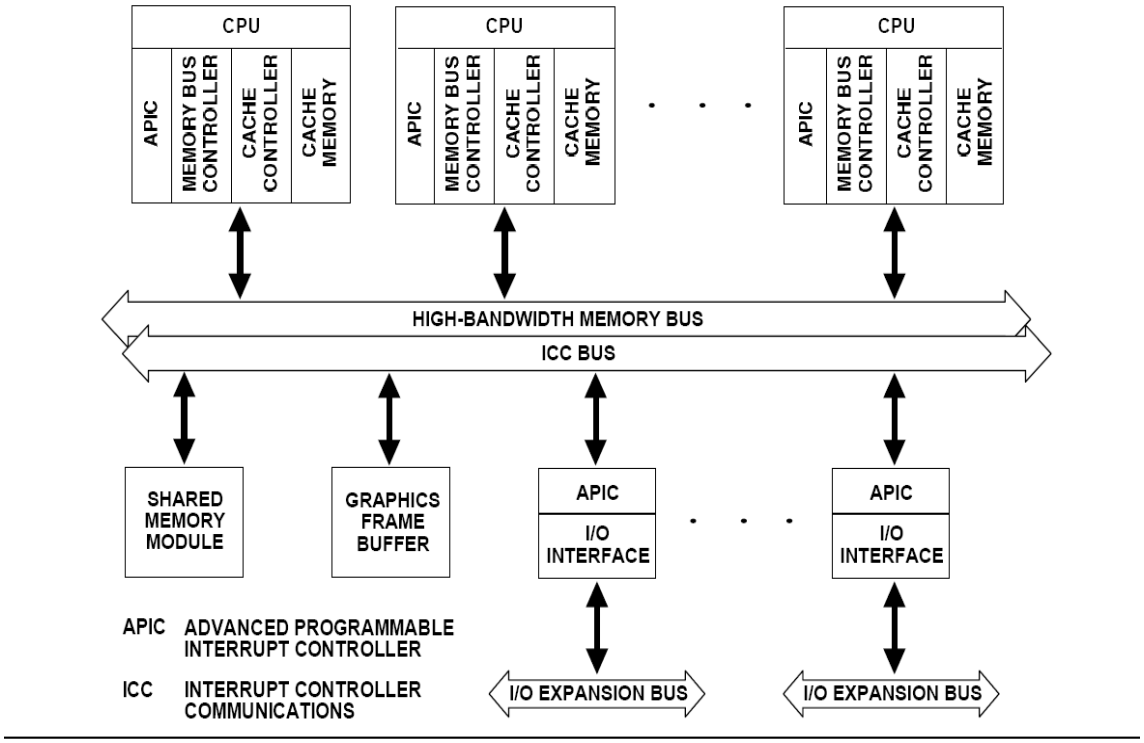


Abbildung 3.1: Architektur eines Multi-Prozessor-Systems ([1] Seite 2-2)

cation-Prozessoren (APs). Abbildung 3.2 zeigt ein Beispielsystem. Welcher Prozessor als BSP verwendet wird, wird von der Hardware in Zusammenhang mit dem BIOS festgelegt. Diese Unterscheidung spielt nur beim Hochfahren und beim Herunterfahren des Systems eine Rolle, die Aufgabe des BSPs ist es, das Betriebssystem zu booten, und danach die APs zu aktivieren. Sobald die APs arbeiten, werden alle Prozessoren gleichberechtigt behandelt.

Alle Prozessoren, d.h. sowohl der BSP als auch die APs, werden aus Kompatibilitätsgründen im eingeschränkten *Real Mode* gestartet. Beim Hochfahren muss jeder Prozessor in den leistungsfähigeren *Protected Mode* wechseln.

3.1.2 *Advanced Programmable Interrupt Controller* (APICs)

Die Möglichkeiten, die der *Programmable Interrupt Controller* (PIC), der in einem Mono-Prozessor-System verwendet wird, bietet, reichen in einem Multi-Prozessor-System nicht, da man in der Lage sein möchte, Interrupts beliebig auf alle Prozessoren zu verteilen. Deswegen wurden APICs eingeführt. Die APIC-Architektur basiert auf zwei grundlegenden Elementen, den lokalen APICs und den I/O-APICs, die über den ICC-Bus miteinander kommunizieren und gemeinsam für die Verteilung der Interrupts verantwortlich sind. Da die APICs mit den früher verwendeten PICs kooperieren, ist die Abwärtskompatibilität weiterhin gewährleistet.

Der lokale APIC übernimmt zwei wichtige Aufgaben, er nimmt Interrupts sowohl von lokalen Quellen als auch von einem I/O-APIC entgegen und leitet diese an den

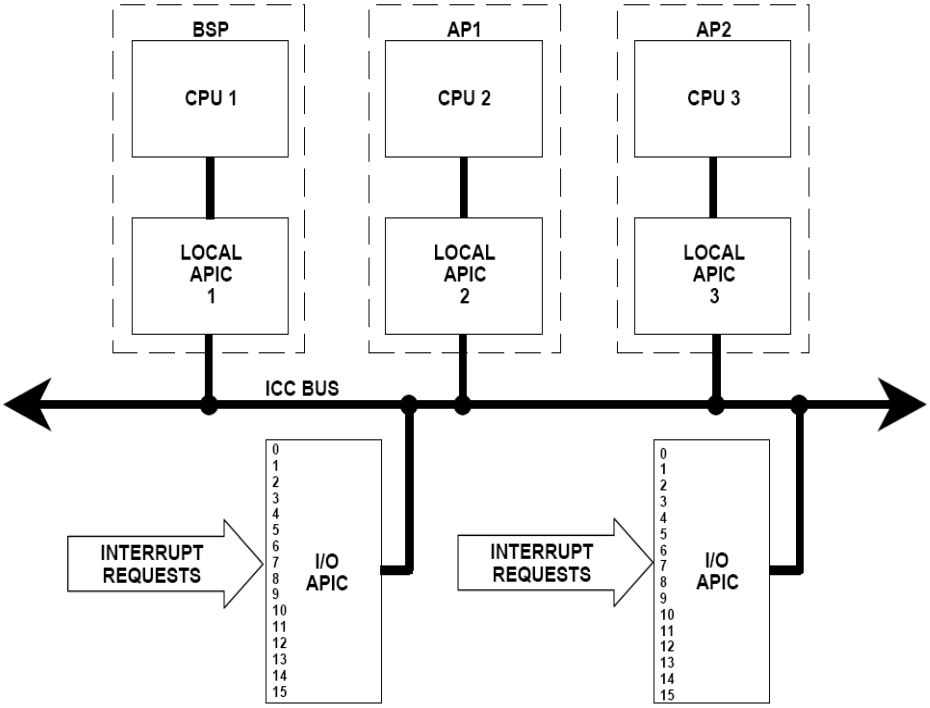


Abbildung 3.2: *Bootstrap.*- und *Application*-Prozessoren ([1] Seite 2-3)

Prozessor weiter. Zusätzlich ist er in der Lage, Inter-Prozessor-Interrupts (IPIs) zu jedem anderen lokalen APIC zu schicken. Inter-Prozessor-Interrupts ermöglichen die Kommunikation der Prozessoren untereinander.

Die Hauptaufgabe des I/O-APICs besteht darin, Interrupts von angeschlossenen I/O-Geräten entgegen zu nehmen, und diese in Form von Interrupt-Nachrichten an einen lokalen APIC weiter zu leiten. In einem Multi-Prozessor-System ist der I/O-APIC in der Lage, Interrupts nach Belieben auf alle lokalen APICs zu verteilen.

Die APICs arbeiten in einem der drei folgenden Interrupt-Modi:

1. PIC-Modus: Die APIC Architektur wird übergangen, und das System dazu gezwungen, im Mono-Prozessor-Modus zu arbeiten (siehe unten).
2. *Virtual Wire*-Modus: Die APICs werden als „virtuelle Kabel“ verwendet, ansonsten arbeitet das System genau wie im PIC-Modus (siehe unten).
3. *Symmetric I/O*-Modus: In diesem Modus werden die APICs genutzt und somit der Multi-Prozessor-Betrieb ermöglicht (siehe unten).

Die ersten beiden Modi garantieren die Abwärtskompatibilität, da die APICs in diesen Modi gar nicht oder nur zur Weiterleitung der Interrupts des PICs verwendet werden. Mindestens einer dieser beiden Modi muss in jedem Multi-Prozessor-System implementiert sein, da das Betriebssystem in einem dieser beiden Modi gebootet wird. In welchem kann aus den *MP Configuration Bytes* auslesen (siehe Abschnitt

6.1.1, Tabelle 6.2). Später wird dann gegebenenfalls in den *Symmetric I/O*-Modus gewechselt.

PIC-Modus

Der PIC-Modus ist identisch mit früheren Systemen, die APICs werden komplett übergangen, und Interrupts werden ausschließlich vom PIC behandelt. Abbildung 3.3 zeigt die Arbeitsweise des PIC-Modus und wie das *Interrupt Mode Configuration Register* (IMCR) verwendet wird, um die APICs zu umgehen. In diesem Modus werden alle Interrupts an den BSP geschickt. Um in den *Symmetric I/O*-Modus umzuschalten, muss das IMCR auf 01h gesetzt werden.

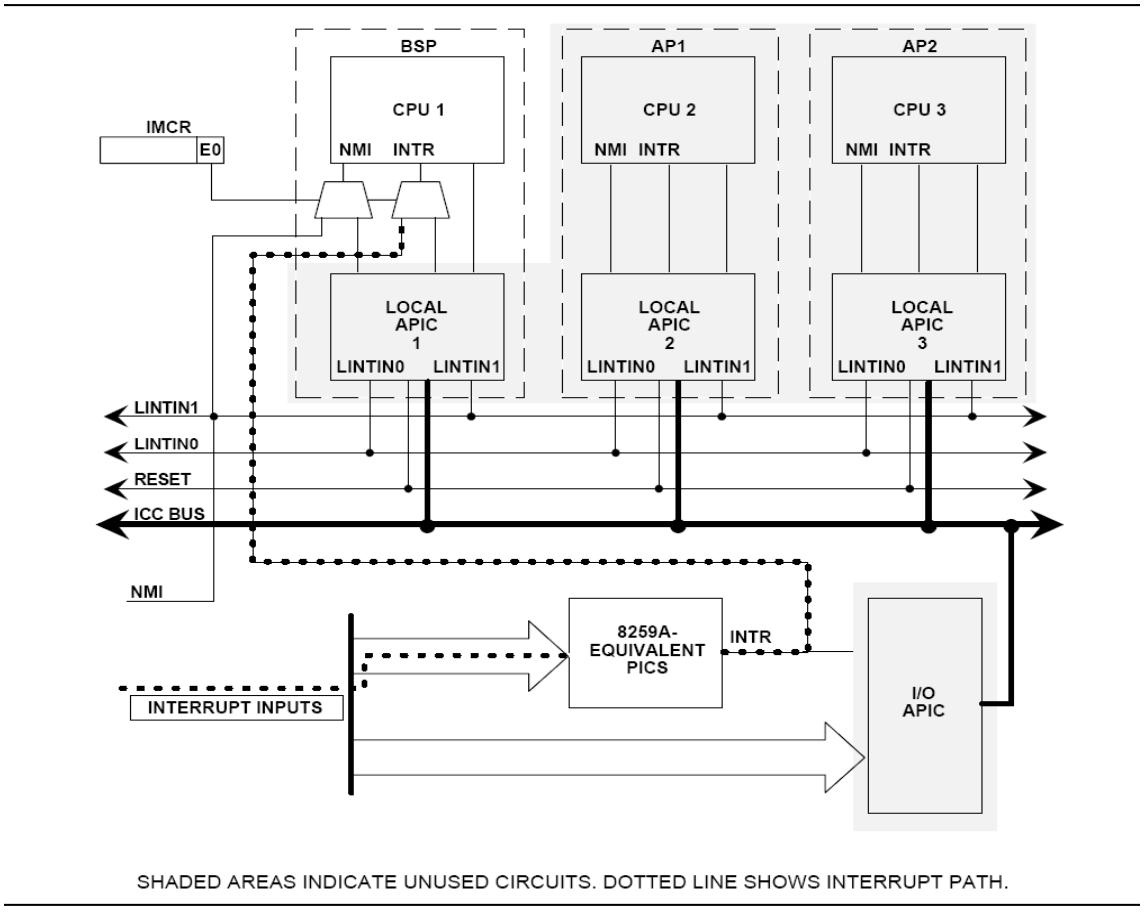


Abbildung 3.3: PIC-Modus ([1] Seite 3-8)

Der Zugriff auf das IMCR wird über zwei I/O-Ports ermöglicht. Um das IMCR auszuwählen, muss der Wert 70h auf den Port 22h geschrieben werden, danach muss der neue Wert für das IMCR auf den Port 23h geschrieben werden. Der *Power On Default*-Wert des IMCR ist 00h. Um den PIC-Modus zu verlassen und Interrupts durch den APIC zu schicken, muss das IMCR auf 01h gesetzt werden.

Bei einem Reset muss das IMCR wieder zurückgesetzt werden, um den PIC-Modus für den Systemstart zu aktivieren.

Sollte das System im *Virtual Wire*-Modus starten und der PIC-Modus nicht implementiert sein, ist das IMCR optional und sollte vom Betriebssystem nicht beschrieben werden.

Virtual Wire-Modus

Im *Virtual Wire*-Modus werden Interrupts genau wie im PIC-Modus vom PIC behandelt und immer an den BSP geschickt. Der Unterschied zum PIC-Modus liegt darin, dass die APICs als „virtuelle Kabel“ verwendet werden und jeder Interrupt über die APICs zum BSP weitergeleitet wird. Es gibt zwei Varianten des *Virtual Wire*-Modus, in der ersten (siehe Abbildung 3.4) wird ein Interrupt nur durch den lokalen APIC des BSP geschickt, und der I/O-APIC wird nicht verwendet, in der zweiten (siehe Abbildung 3.5) macht ein Interrupt den Umweg über den I/O-APIC und den lokalen APIC.

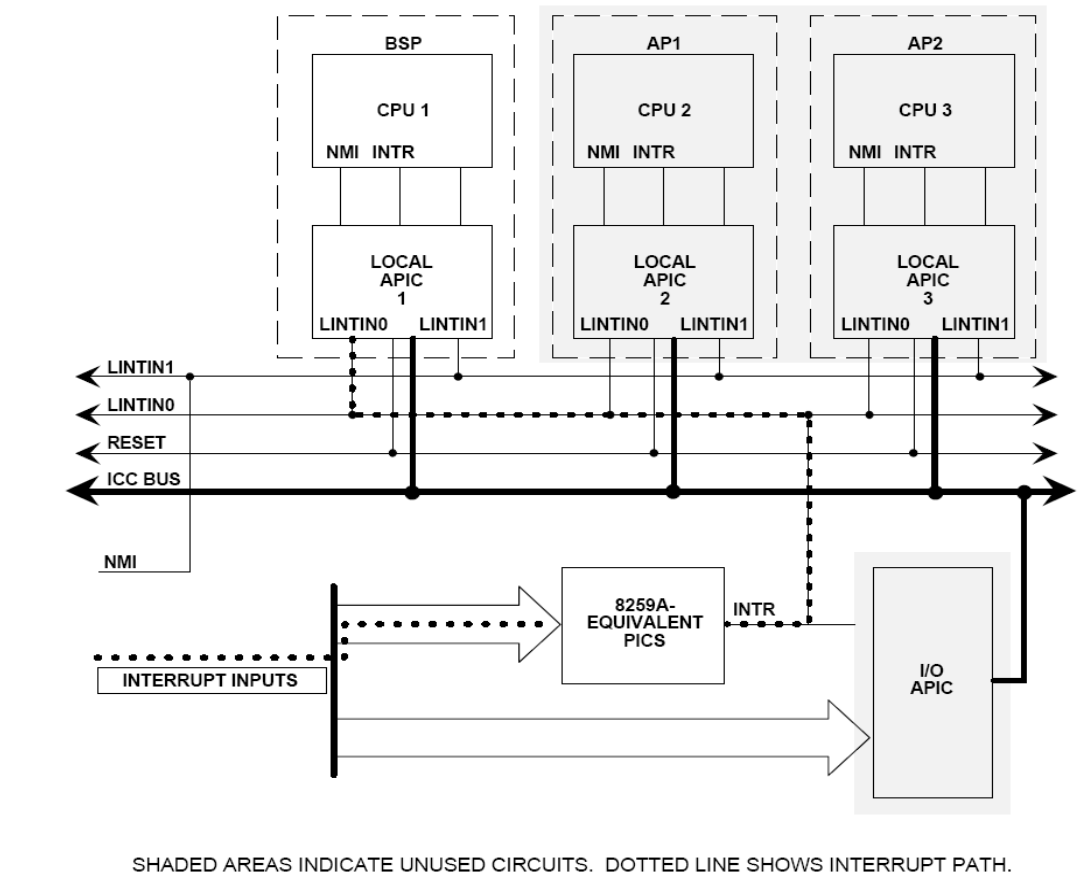


Abbildung 3.4: *Virtual Wire*-Modus ohne I/O-APIC ([1] Seite 3-9)

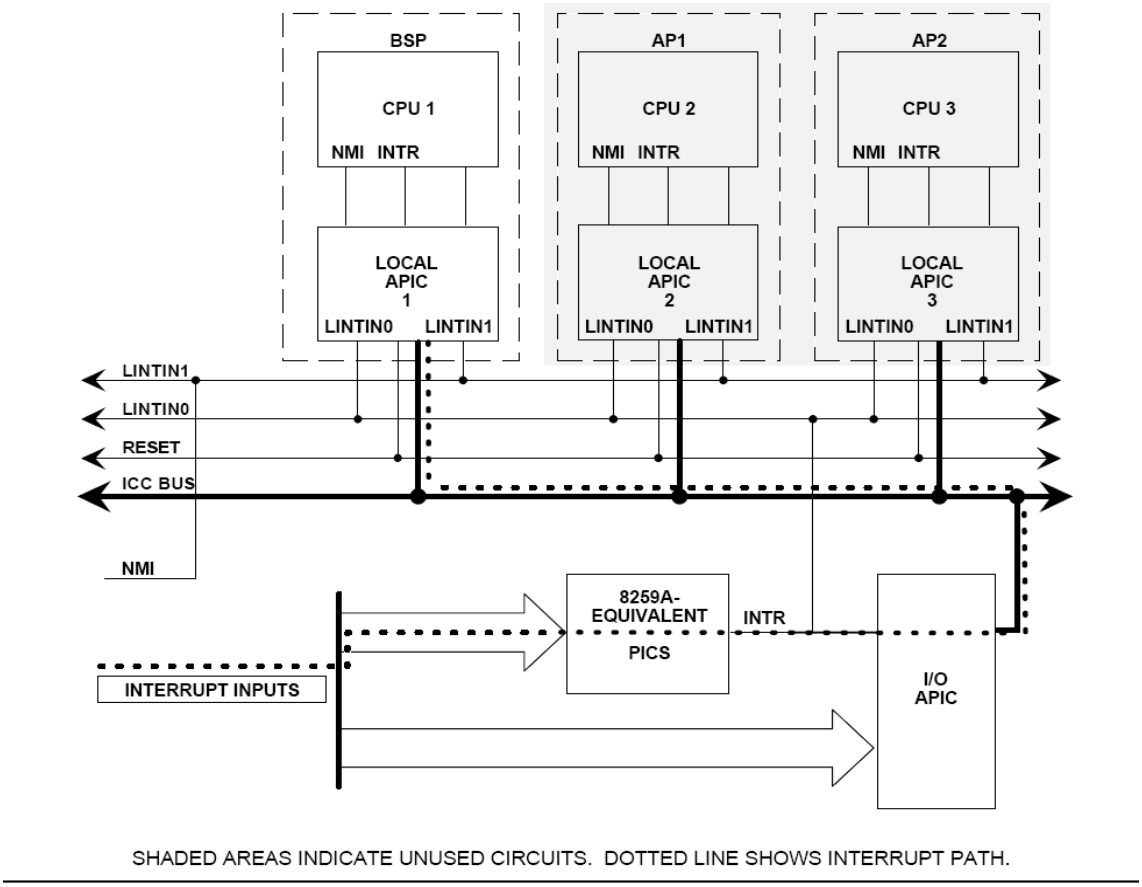


Abbildung 3.5: *Virtual Wire*-Modus mit I/O-APIC ([1] Seite 3-10)

Symmetric I/O-Modus

Multi-Prozessor-Betriebssysteme arbeiten im *Symmetric I/O*-Modus. In diesem Modus werden Interrupts vom I/O-APIC in Form von Interrupt-Nachrichten an die lokalen APICs der einzelnen Prozessoren weitergeleitet. Die Interrupts des PICs sind entweder komplett ausmaskiert, oder der PIC gibt seine Interrupts an den I/O-APIC weiter, der sie dann nach eigenem Belieben auf die lokalen APICs verteilt. Abbildung 3.6 zeigt den *Symmetric I/O*-Modus.

Sollte das System im *Virtual Wire*-Modus gebootet werden, und das IMCR ist nicht vorhanden, ist außer der Programmierung des I/O-APICs keine weitere Aktion nötig, um in den Symmetric-I/O-Modus umzuschalten.

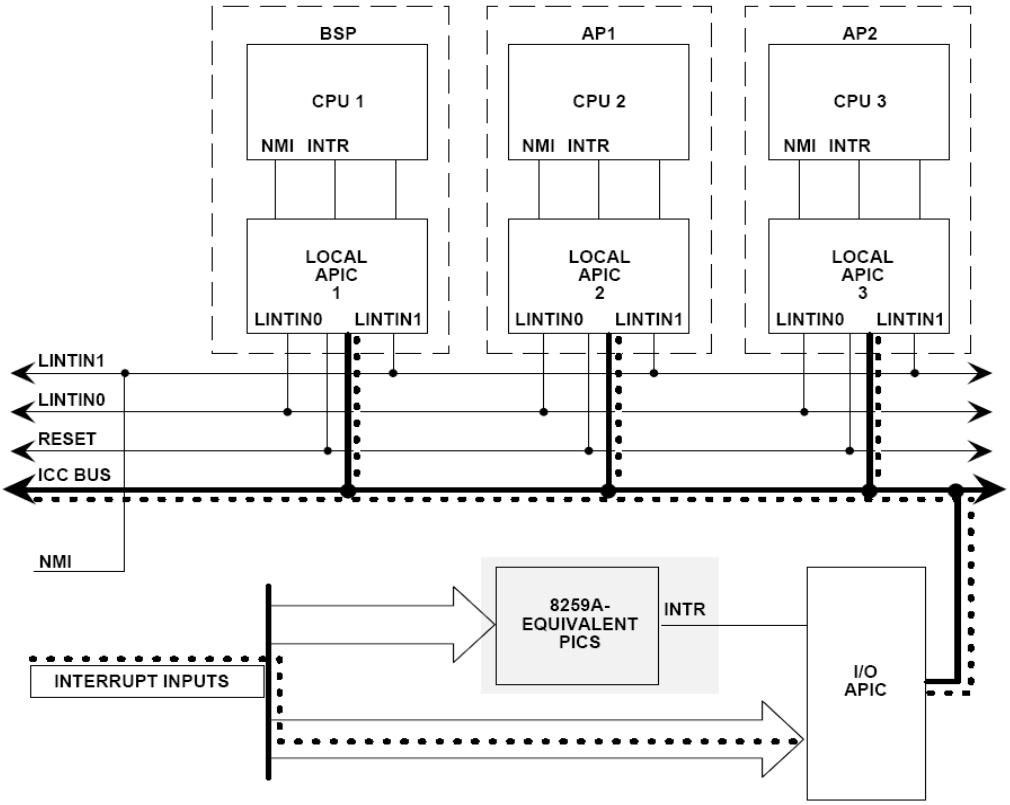


Abbildung 3.6: *Symmetric I/O*-Mode ([1] Seite 3-11)

3.1.3 Systemspeicher

Der Adressraum in einem 32-Bit System enthält vier Gigabyte. Speicher-gemappte I/O-Geräte sollten ans obere Ende des Speichers gemappt werden. Die Standard-Basis-Adressen der APICs sind FEC00000h und FEE00000h. Abbildung 3.7 zeigt einen Ausschnitt aus dem Adressraum.

Der für den lokalen APIC reservierte Adressraum wird von jedem Prozessor verwendet, um den eigenen lokalen APIC zu adressieren. Der für den I/O-APIC reservierte Adressraum wird von allen Prozessoren verwendet, um den I/O-APIC zu adressieren.

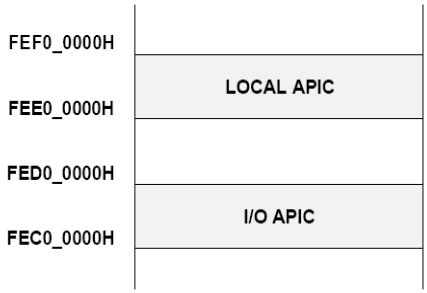


Abbildung 3.7: Ausschnitt aus dem Adressraum mit APICs ([1] Seite 3-2)

3.2 Speichermanagement

Wie in Abschnitt 3.1.1 angesprochen wird aus Kompatibilitätsgründen jeder Prozessor im *Real Mode* gestartet. Aktuelle Prozessoren sind in diesem Modus funktional stark eingeschränkt, weshalb sie beim System Start möglichst schnell in den leistungsfähigeren *Protected Mode* wechseln. In beiden Modi ist der Speicher in Segmente aufgeteilt, die Summe aller Segmente entspricht dem linearen Adressraum. Lineare Adressen werden gebildet, indem ein Segment adressiert wird (über die Segmentregister CS, DS, ES, GS, FS oder SS) und auf diese Segmentadresse ein Offset addiert wird. Sollte kein *Paging* verwendet werden (wie es in MPStuBS der Fall ist), entspricht der lineare Adressraum dem physikalischen Adressraum, und lineare Adressen können direkt verwendet werden, um Speicherstellen zu adressieren. Im Folgenden wird genauer auf das Speichermanagement im *Real* und im *Protected Mode*, wie in [5] beschrieben, eingegangen.

3.2.1 *Real Mode*

Im *Real Mode* ist der Speicher in 16 Byte große Segmente aufgeteilt. Jedes Segmentregister beinhaltet eine 16 Bit Adresse, die mit 16 multipliziert wird um, die Segmentadresse zu erhalten. Um eine lineare Adresse zu erzeugen, wird auf diese Segmentadresse der 16 Bit breite Offset aufaddiert. Somit ergibt sich eine 20 Bit breite lineare Adresse. Damit ist ein Megabyte Speicher adressierbar ($2^{20} = 1.048.576 = 1MB$). Abbildung 3.8 zeigt die Adressumsetzung im Real Mode.

Da in einem 32 Bit System die vollen vier Gigabyte Adressraum adressierbar sein sollen, um z.B. die APICs adressieren zu können (siehe Abschnitt 3.1.3), muss im *Protected Mode* eine andere Form der Adressumsetzung realisiert sein.

3.2.2 *Protected Mode*

Im *Protected Mode* haben die Segmente eine variable Größe. Einzelnen Programmen können eigene Segmente zugewiesen werden, so dass mehrere Programme auf einem

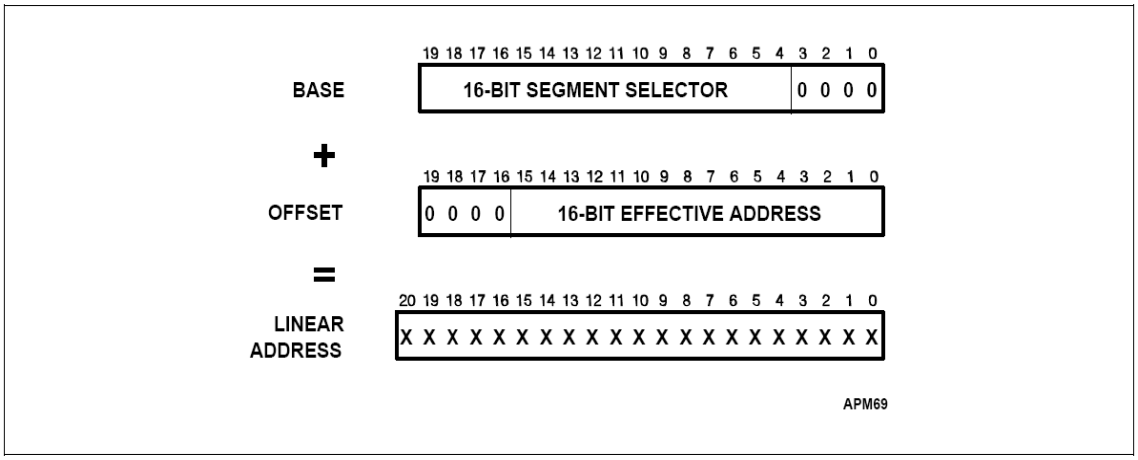


Abbildung 3.8: Adressumsetzung im *Real Mode* ([5] Seite 9-2)

Prozessor laufen können, ohne sich gegenseitig zu behindern. In der *Global Descriptor Table* (GDT) befindet sich für jedes Segment ein Segmentdeskriptor, der unter anderem den Typ, die Anfangsadresse und die Größe eines Segments beschreibt. Die Anfangsadresse des GDT steht im *Global Descriptor Table Register* (GDTR).

Um ein Byte in einem Segment zu adressieren, muss aus einer logischen Adresse eine lineare Adresse erzeugt werden. Eine logische Adresse besteht aus einem 16 Bit Segmentselektor und einem 32 Bit Offset. Der Segmentselektor befindet sich im entsprechenden Segmentregister und stellt unter anderem einen Index für eine *Descriptor Table* (in unserem Fall die GDT) zur Verfügung, über den auf den entsprechenden Segmentdeskriptor zugegriffen werden kann. Um eine lineare Adresse zu erhalten, wird aus dem gefundenen Segmentdeskriptor die Anfangsadresse des Segments ausgelesen, und das Offset der logischen Adresse aufaddiert. Abbildung 3.9 zeigt die Adressumsetzung im *Protected Mode*.

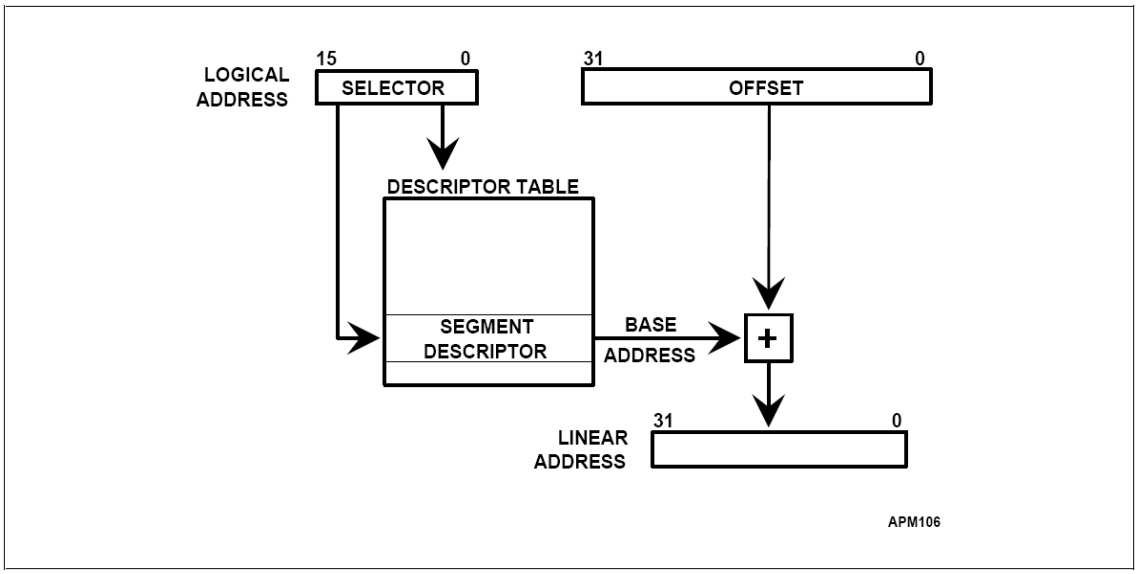


Abbildung 3.9: Adressumsetzung im *Protected Mode* ([5] Seite 11-9)

Die Abbildungen 3.10 und 3.11 zeigen einen Segmentselektor und einen Segmentdeskriptor.

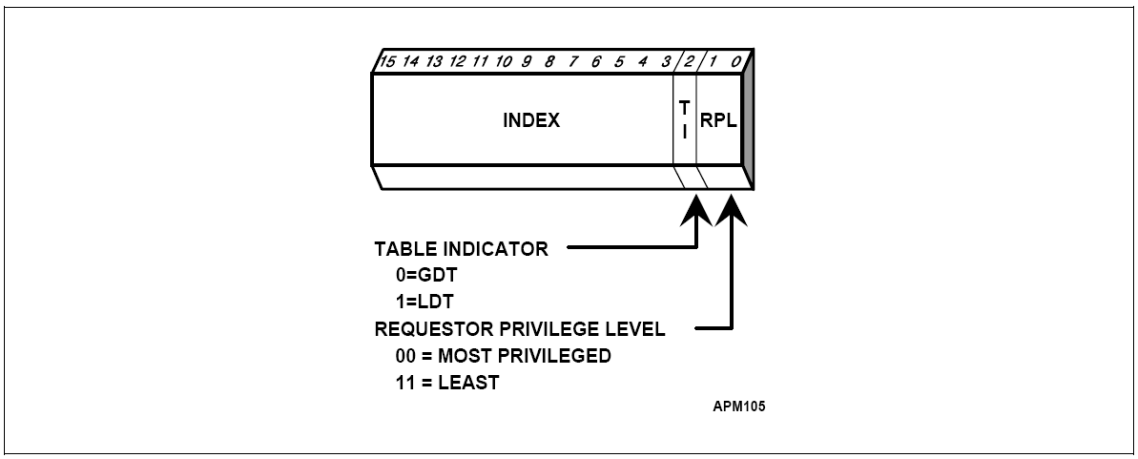


Abbildung 3.10: Segmentselektor ([5] Seite 11-10)

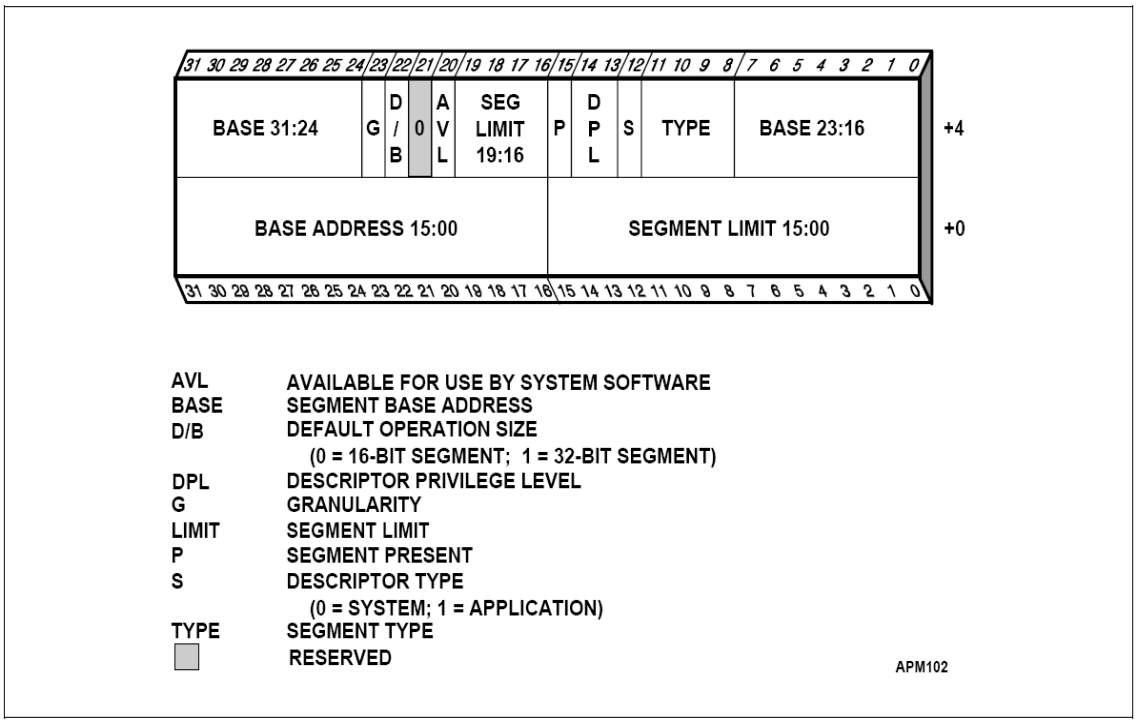


Abbildung 3.11: Segmentdeskriptor ([5] Seite 11-12)

3.3 BIOS-Überblick

Das BIOS wird unmittelbar nach jedem Einschalten des PCs gestartet. Es bringt das System in seinen initialen Zustand und leitet das Booten des Betriebssystems ein. Das BIOS hat folgende Aufgaben [1]:

In einem Mono-Prozessor-System:

1. Testen der System-Komponenten.
2. Anlegen von Konfigurations-Tabellen, die das Betriebssystem mit Informationen über die Hardware-Konfiguration versorgen.
3. Initialisieren des Prozessors und der restlichen Systemkomponenten.

Zusätzlich in einem Multi-Prozessor-System:

1. Die APs schlafen legen, damit nur der BSP den BIOS-Code ausführt.
2. Initialisieren der APICs.
3. Anlegen der *MP Floating Pointer Structure* und der *MP Configuration Table* (mehr dazu in Abschnitt 6.1), die das Betriebssystem mit den nötigen Informationen über die Konfiguration der APICs und der anderen MP-Komponenten versorgen.

3.3.1 BIOS POST Initialisierung

Nach jedem Einschalten des PCs und nach jedem Drücken des Reset-Knopfes wird von der Hardware ein systemweites Reset eingeleitet, das alle Systemkomponenten in einen initialen Zustand versetzt. Danach beginnen alle Prozessoren mit der Ausführung der POST (*Power On Self Test*) Prozedur, die alle Systemkomponenten in einen bekannten Startzustand versetzt und Systemtabellen für das Betriebssystem anlegt.

3.3.2 Prozessorstart

Nach einem systemweiten Reset muss dafür gesorgt werden, dass nur ein Prozessor das BIOS ausführt, dies kann von der Hardware oder vom BIOS garantiert werden [1]. Sollte das BIOS dafür verantwortlich sein, hat es die Aufgabe, den BSP zu bestimmen und alle anderen Prozessoren nach dem POST schlafen zu legen. Die APIC ID kann verwendet werden, um jeden Prozessor zu identifizieren. Nur der BSP fährt dann nach dem POST mit dem Laden des Betriebssystems fort.

3.3.3 Programmierung der APICs

Ob ein APIC vom BIOS programmiert werden muss, hängt davon ab, in welchem Interrupt-Modus das System startet. Sollte der Standard-Modus der PIC-Modus sein, ist keine Programmierung der APICs nötig, ist es aber der *Virtual Wire*-Modus, muss mindestens der lokale APIC des BSP (eventuell auch der I/O-APIC, je nachdem welcher *Virtual Wire*-Modus verwendet wird) so programmiert werden, dass er als „virtuelles Kabel“ fungiert, da jede Mono-Prozessor-Software ohne weitere Konfiguration der APICs fehlerfrei laufen muss.

3.3.4 Initialer Systemzustand

Der initiale Systemzustand ist der Zustand, in dem das BIOS das System an das Betriebssystem übergibt. Die Multi-Prozessor-Komponenten befinden sich, wenn alles fehlerfrei gelaufen ist, in folgendem Zustand [1]:

- Alle lokalen APICs sind ausgeschaltet, mit Ausnahme des lokalen APICs des BSP, falls das System im *Virtual Wire*-Modus gestartet wurde.
- Sollte das IMCR vorhanden sein, steht es, je nachdem in welchem Interrupt-Modus das System gestartet wurde, auf 0 oder auf 1.
- Alle Prozessoren befinden sich im 16-Bit *Real Mode*.
- Alle APs befinden sich im HALT-Zustand.

Das BIOS ist darüberhinaus dafür verantwortlich, dass alle Interrupts ausgeschaltet sind, bevor es die Kontrolle an das Betriebssystem übergibt.

3.4 Zusammenfassung

Beim Systemstart eines Multi-Prozessor-Systems wird einer der Prozessoren als BSP ausgewählt. Dieser Prozessor startet das Betriebssystem, und weckt die anderen Prozessoren. Alle Prozessoren befinden sich zunächst im eingeschränkten *Real Mode* und wechseln dann in den leistungsfähigeren Protected Mode.

Die APICs befinden sich nach dem Systemstart entweder im PIC- oder im *Virtual Wire*-Modus. In diesen Modi werden alle Interrupts an den BSP geschickt. Um die Interrupts auf mehrere Prozessoren zu verteilen, müssen die APICs im *Symmetric I/O*-Modus betrieben werden.

Bevor auf den Entwurf von MPStuBS eingegangen werden kann, werden im folgenden Kapitel grundlegende Konzepte eines Betriebssystems vorgestellt.

Kapitel 4

Betriebssystem-Kern-Konzepte

Das Lehrbetriebssystem OOStuBS implementiert einen Kern für ein Mono-Prozessor-System. In diesem Kapitel werden grundlegende Konzepte eines solchen Kerns vorgestellt, die auch in MPStuBS verwendet werden sollen. Es wird auf Probleme eingegangen, die diese Konzepte mit sich bringen, wenn sie in einem Multi-Prozessor-System eingesetzt werden, und wie diese Probleme gelöst werden.

4.1 Einführung

In einem Computersystem wird auf zwei verschiedenen Ebenen gearbeitet, der Benutzer Ebene und der Kern Ebene. Die Kernebene wird durch das Betriebssystem realisiert. Abbildung 4.1 zeigt diese Aufteilung.

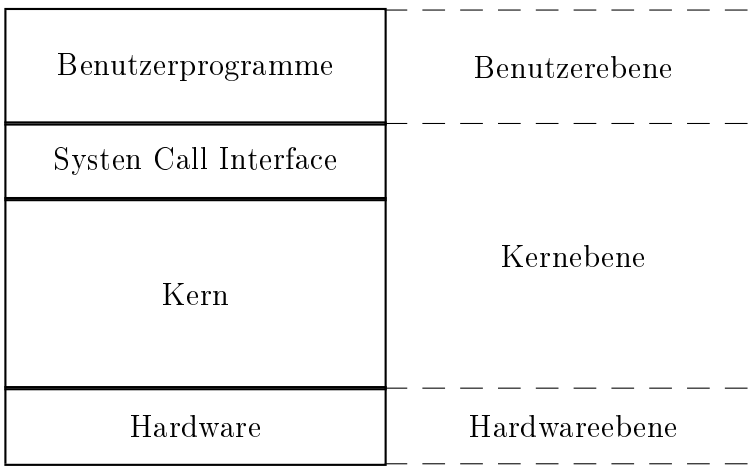


Abbildung 4.1: Ebenen-Modell (nach [10])

Der Kern repräsentiert die Schnittstelle zur Hardware, d.h. er programmiert und kontrolliert die angeschlossenen Geräte. Er arbeitet in einem privilegierten Modus, der es ihm ermöglicht, privilegierte Operationen (z.B. Hardware-Zugriffe) auszuführen. Benutzerprogramme, die solche Operationen ausführen wollen, müssen den

Kern darum bitten. Dafür stellt er eine Schnittstelle zur Verfügung, die sogenannten System Calls.

Auf Benutzerebene wird im unprivilegierten Modus gearbeitet. Dies ist ein eingeschränkter Modus, in dem privilegierte Operationen nicht ausgeführt werden können. Wenn ein Benutzerprogramm privilegierte Operationen durchführen möchte, führt es einen entsprechenden *System Call* aus, der den Kern veranlasst, die gewünschte Operation durchzuführen. Somit hat der Kern die Kontrolle über alle Benutzerprogramme, und kann verhindern, dass parallel laufende Benutzerprogramme sich gegenseitig behindern.

Die Benutzerebene und die Kernebene werden auch als Kontrollflussebenen bezeichnet. Im Folgenden wird das Kontrollflussebenen-Modell vorgestellt.

4.2 Kontrollflussebenen-Modell

Im Kontrollflussebenen-Modell werden Benutzer- und Betriebssystemaktivitäten in Kontrollflussebenen mit verschiedenen Prioritäten eingeteilt [8]. Diese Aufteilung ermöglicht die Koordination der Aktivitäten, und gibt Ebenen mit hoher Priorität die Kontrolle über Ebenen mit niedriger Priorität.

4.2.1 Kontrollflussebenen-Modell ohne Verdrängung

Man setze voraus, es gibt zwei Kontrollflussebenen, die Anwendungskontrollflussebene E_0 (Priorität 0) und die Unterbrechungskontrollflussebene E_1 (Priorität 1). Ein Kontrollfluss einer bestimmten Ebene kann grundsätzlich nur von einem Kontrollfluss einer Ebene mit höherer Priorität unterbrochen werden. D.h. ein Kontrollfluss der Ebene E_0 kann von einem Kontrollfluss der Ebene E_1 unterbrochen werden. Ein Kontrollfluss der Ebene E_1 kann nicht unterbrochen werden, da in diesem Beispiel keine Ebene mit der Priorität 2 vorliegt (siehe Abbildung 4.2).

Sollte ein Kontrollfluss der Ebene E_1 einen Kontrollfluss der Ebene E_0 unterbrochen haben, und ein zweiter Kontrollfluss der Ebene E_1 signalisiert, dass er Aktivitäten durchführen will, gibt es folgendes Problem: Der zweite Kontrollfluss der Ebene E_1 ist nicht in der Lage, den ersten zu unterbrechen, aber er hat eine höhere Priorität als der ursprünglich unterbrochene Kontrollfluss der Ebene E_0. Deshalb wird der zweite Kontrollfluss der EbeneE_1 in eine Warteschlange eingereiht, die abgearbeitet werden muss, bevor zurück zu Ebene E_0 gewechselt werden kann. Für jede Kontrollflussebene muss eine solche Warteschlange vorliegen. Kontrollflüsse der gleichen Ebene, die gleichzeitig Aktivitäten ausführen wollen, werden also sequentialisiert, d.h sie werden nacheinander komplett abgearbeitet, bevor die Kontrolle an eine andere Ebene abgegeben wird (*run to completion*-Semantik, siehe Abbildung 4.3).

Unabhängig von der Anzahl der Kontrollflussebenen gilt also immer, dass Kontrollflüsse nur von Kontrollflüssen höherer Priorität unterbrochen werden können, und dass Kontrollflüsse gleicher Priorität sequentialisiert werden.

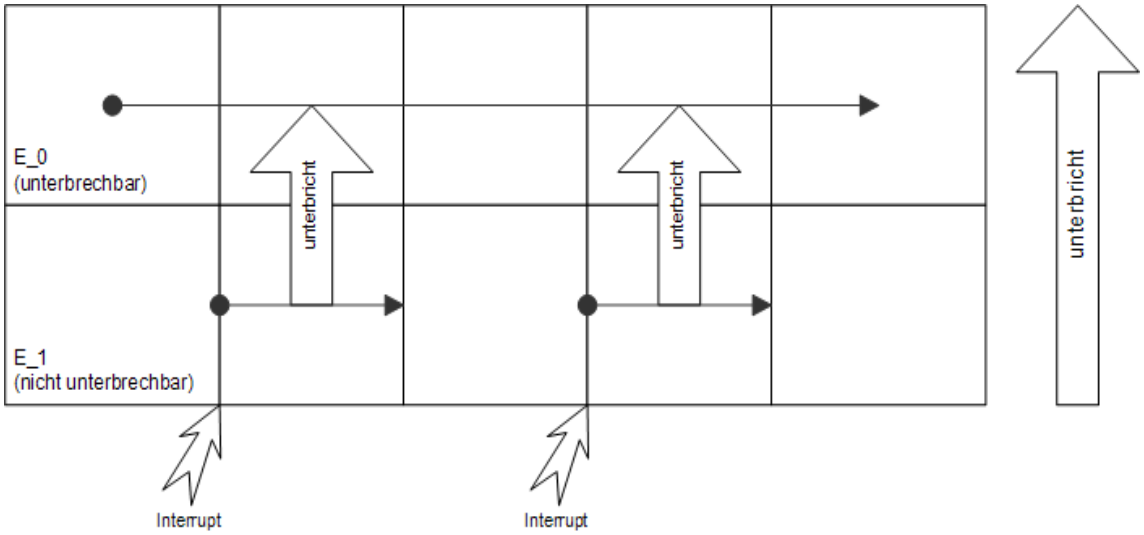


Abbildung 4.2: Kontrollflussebenen (nach [8])

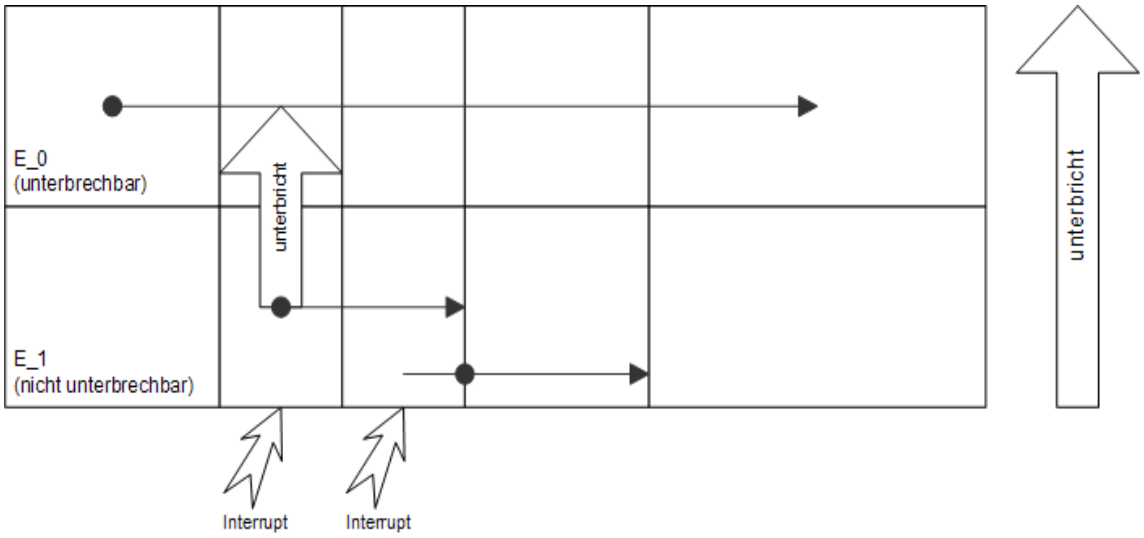


Abbildung 4.3: Sequentialität in einer Kontrollflussebene (nach [8])

Konsistenzsicherung

Die Daten, die ein Kontrollfluss verwendet, sind ebenfalls einer Ebene zugeordnet, normalerweise der gleichen Ebene wie der Kontrollfluss selbst. Da innerhalb einer Ebene alle Aktivitäten sequentialisiert werden, treten Konsistenzprobleme nur auf, wenn aus einer anderen Ebene auf diese Daten zugegriffen wird. Sollte also z.B. ein Kontrollfluss der Ebene E_0 inmitten einer Aktualisierung einer Datenstruktur von einem Kontrollfluss der Ebene E_1 unterbrochen werden, der auf der gleichen Datenstruktur arbeitet, kann es zu Inkonsistenzen kommen. Dieses Problem kann gelöst werden, indem der Kontrollfluss der Ebene E_0 für die Dauer der Aktualisierung auf die höchstpriore Ebene, aus der auf die entsprechende Datenstruktur zugegriffen wird, wechselt.

4.2.2 Kontrollflussebenen-Modell mit Verdrängung

Geht man davon aus, dass alle Benutzerprogramme in einem Betriebssystem auf der gleichen Ebene ablaufen, macht das Kontrollflussebenen-Modell ohne Verdrängung keinen Sinn, da die Programme dann nicht parallel laufen könnten. Deshalb lässt man zu, dass auf dieser Ebene Kontrollflüsse verdrängt werden können.

Konsistenzsicherung

Da die rein sequentielle Ausführung von Kontrollflüssen innerhalb einer Ebene, die das Verdrängen von Kontrollflüssen zulässt, nicht mehr gegeben ist, reicht der oben genannte Mechanismus nicht mehr, um die Datenkonsistenz zu gewährleisten. Sollte z.B. ein Kontrollfluss inmitten einer Aktualisierung einer Datenstruktur von einem anderen verdrängt werden, der auf die gleiche Datenstruktur zugreift, kann es zu Inkonsistenzen kommen. Ist das Verdrängen von Kontrollflüssen innerhalb einer Ebene erlaubt, muss die Konsistenz der Daten dieser Ebene zusätzlich über gegenseitigen Ausschluss gesichert werden.

4.3 Gegenseitiger Ausschluss

Die Hardware garantiert, dass einzelne Zugriffe auf den Speicher nicht von anderen Zugriffen unterbrochen werden können. Das Aktualisieren einer Datenstruktur erfordert allerdings mehrere aufeinander folgende Zugriffe, deren Reihenfolge strikt eingehalten werden muss. Sollte ein zweiter Kontrollfluss gleichzeitg die gleiche Datenstruktur beschreiben, wird diese Reihenfolge durcheinander gebracht, und es kann zu Inkonsistenzen kommen. Solche Wettbewerbssituationen werden *Race Conditions* genannt.

Jede beliebige Sequenz von Befehlen, die auf einer von mehreren Kontrollflüssen verwendeten Datenstruktur arbeitet, kann zu *Race Conditions* führen. Man bezeichnet eine solche Sequenz als kritischen Bereich. Um das Auftreten von *Race Conditions* zu verhindern, muss gewährleistet sein, dass immer nur ein Kontrollfluss einen kritischen Bereich abarbeitet. Dieses Konzept wird gegenseitiger Ausschluss genannt.

Eine beliebte Implementierung des gegenseitigen Ausschlusses sind Semaphoren. Nach der Definition von Dijkstra ist ein Semaphor eine ganze Zahl, die die Werte 0 und 1 annehmen kann. 0 bedeutet, dass der kritische Bereich frei ist, 1 dass er belegt ist. Der Wert eines Semaphors kann über die P- und die V-Operation verändert werden. P wird beim Betreten des kritischen Bereichs ausgeführt, und dekrementiert den Wert des Semaphors falls dieser 1 ist, um zu signalisieren, dass der kritische Bereich jetzt belegt ist. Sollte der Wert auf 0 stehen, legt P den ausführenden Kontrollfluss schlafen, da der kritische Bereich bereits von einem anderen Kontrollfluss belegt ist. V wird beim Verlassen des kritischen Bereichs ausgeführt, und weckt gegebenenfalls einen schlafenden Kontrollfluss, der den kritischen Bereich betreten will. Sollte kein Kontrollfluss auf den kritischen Bereich warten, inkremen-

tiert V den Wert des Semaphors, und signalisiert damit, dass der kritische Bereich wieder frei ist.

Das Ausführen von P und V muss auf einer Ebene ablaufen, auf der der Kontrollfluss nicht verdrängt werden kann, da sonst die Gefahr besteht, dass ein Kontrollfluss, der gerade den Wert des Semaphors gelesen hat, verdrängt wird, bevor er ihn inkrementieren kann. Das führt dazu, dass womöglich beide Kontrollflüsse den kritischen Bereich betreten und Inkonsistenzen auftreten können.

4.4 Probleme in Multi-Prozessor-Systemen

Die in Abschnitt 4.2 und 4.3 angesprochenen Konzepte führen in Multi-Prozessor-Systemen ohne Erweiterungen zu Problemen. Im Folgenden werden diese Probleme angesprochen, und Lösungsansätze genannt.

4.4.1 Kontrollflussebenen-Modell

Um die Leistungsfähigkeit eines Multi-Prozessor-Systems zu maximieren, müssen alle Prozessoren in der Lage sein, Kontrollflüsse aller Ebenen zu bearbeiten. Es muss allerdings trotzdem gewährleistet bleiben, dass in Ebenen, die das Verdrängen von Kontrollflüssen verbieten, alle Kontrollflüsse nach der *run to completion*-Semantik ausgeführt werden. D.h., sollte ein Prozessor einen Kontrollfluss einer solchen Ebene ausführen wollen, und ein anderer Prozessor arbeitet gerade auf dieser Ebene, muss er damit solange warten, bis der andere Prozessor diese Ebene wieder verlassen hat.

In Ebenen, die das Verdrängen von Kontrollflüssen zulassen, können mehrere Kontrollflüsse tatsächlich auf mehreren Prozessoren parallel laufen, solange die verwendeten Mechanismen des gegenseitigen Ausschlusses weiterhin funktionsfähig sind.

4.4.2 Gegenseitiger Ausschluss

Bei gegenseitigem Ausschluss treten nur Probleme auf, wenn mehrere Prozessoren gleichzeitig auf den Synchronisationsmechanismus, z.B. den Semaphor, zugreifen können. Wird dieser Zugriff in einer Ebene ausgeführt, die rein sequentiell abläuft, funktioniert gegenseitiger Ausschluss auch auf einem Multi-Prozessor-System.

4.4.3 Lösungsansätze

Da das gegenseitige Ausschließen direkt abhängig von der Funktionsfähigkeit des Kontrollflussebenen-Modells ist, reicht es, dieses Modell zu erweitern, um auch gegenseitigen Ausschluss zu ermöglichen. Es muss also gewährleistet werden, dass in einer Ebene, die das Verdrängen von Kontrollflüssen verbietet, auch auf Multi-Prozessor-Systemen nur sequentiell gearbeitet wird. Das kann erreicht werden, indem

ein Prozessor, der feststellt, dass ein anderer Prozessor bereits auf der entsprechenden Ebene arbeitet, solange aktiv wartet, bis dieser Prozessor die Ebene wieder verlassen hat. Dafür werden *Spinlocks* verwendet [7].

4.5 *Spinlocks*

Ein *Spinlock* ist ein Synchronisationsmechanismus, der gut geeignet ist, um kurze kritische Bereiche zu synchronisieren. Bevor der kritische Bereich betreten wird, wird das *Spinlock* geschlossen, und mit dem Verlassen des kritischen Bereichs wird es wieder freigegeben. Ein Kontrollfluss, der ein geschlossenes *Spinlock* auffindet, bleibt solange in einer Schleife, bis ein anderer Kontrollfluss das Schloss wieder freigegeben hat. Dann schließt er es, und andere Kontrollflüsse müssen gegebenenfalls warten.

Ein *Spinlock* wird durch ein einziges Bit repräsentiert, dessen Wert beschreibt, ob das Schloss geschlossen oder frei ist. Um zu verhindern, dass mehrere Prozessoren gleichzeitig das *Spinlock* schließen, wird diese Operation atomar ausgeführt, d.h. dass die Hardware garantiert, dass nur ein Prozessor auf die entsprechende Speicherstelle zugreifen kann. Sollten zwei Prozessoren gleichzeitig ein *Spinlock* schließen wollen, werden die Anfragen also von der Hardware sequentialisiert, und einer der beiden Prozessoren findet das Schloss geschlossen vor.

Da es in einem Mono-Prozessor-System nicht vorkommen kann, dass mehrere Prozessoren gleichzeitig auf eine Speicherstelle zugreifen, werden *Spinlocks* in Mono-Prozessor-Systemen nicht verwendet. *Spinlocks* sind also ein effektiver Synchronisationsmechanismus, um in Multi-Prozessor-Systemen kurze kritische Bereiche zu schützen. Sie sollten nicht eingesetzt werden, um lange kritische Bereiche zu synchronisieren, da der wartende Prozessor keine sinnvolle Arbeit ausführen kann, während er auf ein *Spinlock* wartet.

In Multi-Prozessor-Systemen muss das Betreten einer Ebene, die rein sequentiell arbeitet, also mit einem *Spinlock* geschützt werden. Dann arbeiten auf dieser Ebene auch mehrere Prozessoren rein sequentiell.

4.6 Zusammenfassung

Das Kontrollflussebenen-Modell wird verwendet, um das Betriebssystem in einzelne Ebenen zu unterteilen. Diese Ebenen arbeiten entweder mit Verdrängung, um alle Kontrollflüsse einer Ebene gleichberechtig zu behandeln, oder ohne Verdrängung, um sicher zu stellen, dass ein Kontrollfluss seine Arbeit beendet, bevor er die Kontrolle an einen anderen abgibt. Ebenen mit hoher Priorität haben die Kontrolle über Ebenen mit niedriger Priorität.

In einem Multi-Prozessor-System gewährleisten *Spinlocks* die Funktionsfähigkeit des Kontrollflussebenen-Modells.

Im Folgenden werden die Ziele von MPStuBS festgelegt, und besprochen, wie die vorgestellten Konzepte eingesetzt werden können, um diese Ziele zu realisieren. Im letzten Kapitel wird dann detailliert auf die Implementierung von MPStuBS eingegangen.

Kapitel 5

Anforderungen und Ziele

Dieses Kapitel beinhaltet den konzeptionellen Entwurf des SMP-Ports MPStuBS für das Betriebssystem OOStuBS. Grundlegende Probleme beim Entwurf eines Multi-Prozessor-Betriebssystems werden angesprochen, und die verwendeten Konzepte zur Lösung dieser Probleme genannt.

MPStuBS soll ein Betriebssystem für eine *Tightly Coupled*, *Shared Memory*, *Symmetric* Multi-Prozessor-Architektur sein. Die Möglichkeiten, die ein solches System bietet, sollen genutzt werden, d.h. es sollen tatsächlich mehrere Prozesse parallel auf den Prozessoren laufen können. Des Weiteren soll die Verteilung der Aufgaben einer Anwendung für den Anwendungsentwickler transparent sein, und das Betriebssystem MPStuBS soll ebenso wie OOStuBS als Lehrbetriebssystem verwendet werden können. Um diese Ziele zu erreichen, wurden folgende Anforderungen definiert:

- Übersichtlichkeit
- Anforderungen an den Kern
- Anforderungen an den Scheduler
- Synchronisation
- Mono-Prozessor API

5.1 Übersichtlichkeit

OOStuBS wurde ursprünglich als Lehrbetriebssystem entwickelt, d.h. es sollte alle relevanten Teile eines Betriebssystems enthalten, aber trotzdem gut durchstrukturiert, übersichtlich und leicht verständlich sein. Diese Vorgaben wurden so für MPStuBS übernommen, die Erweiterungen am Code sollten also minimal, und die einfache Lesbarkeit weiterhin gewährleistet sein.

5.2 Anforderungen an den Kern

Für die Implementierung eines Multi-Prozessor-Kerns gibt es viele Möglichkeiten. Die einfachste Variante ermöglicht es nur einem Prozessor, den Kern zu betreten, während alle weiteren Prozessoren, die Kernaktivitäten durchführen wollen, warten müssen, bis der erste Prozessor den Kern wieder verlassen hat. Für diese Variante reicht es, den gesamten Kern über einen Synchronisationsmechanismus zu synchronisieren, z.B. indem man den Kern als eine Ebene des Kontrollflussebenen-Modells ohne Verdrängung (siehe Abschnitt 4.2.1) implementiert.

Kompliziertere Varianten ermöglichen es mehreren Prozessoren sich gleichzeitig im Kern zu befinden, dafür wird der Kern in kleinere Module aufgeteilt, die jeweils über einen eigenen Synchronisationsmechanismus geschützt werden. Solange die Prozessoren also nicht das gleiche Modul betreten wollen, können sie sich parallel im Kern befinden.

Für MPStuBS wurde, um die Einfachheit zu gewährleisten, die erste Variante gewählt.

5.3 Anforderungen an den Scheduler

Um die Möglichkeiten eines Multi-Prozessor Systems auszunutzen, müssen alle Prozessoren in der Lage sein, über Scheduleraufrufe Prozesse zu wechseln, neue Prozesse anzumelden, Prozesse zu beenden, oder Prozesse gezielt zu löschen. Dies erfordert Zugriffe auf die globale *Readylist*, in der alle laufbereiten Prozesse warten, die synchronisiert werden müssen.

Um alle Prozesse auf Benutzerebene gleichberechtigt zu behandeln, wird in MPStuBS Zeitscheiben-Scheduling verwendet. Nach jeder abgelaufenen Zeitscheibe wird ein Interrupt generiert, der bei dem unterbrochenen Prozessor einen Prozesswechsel auslöst. Die Interrupts werden gleichmäßig auf alle Prozessoren verteilt.

Sollte der letzte Prozess aus der Readylist beendet werden, und der Scheduler keinen weiteren laufbereiten Prozess finden, muss in eine *Idle-Loop* gesprungen werden. In einem Mono-Prozessor System wird diese *Idle-Loop* im Kontext des letzten aktiven Prozesses ausgeführt. Da dieses Konzept in einem Multi-Prozessor System nicht mehr ausreicht, gibt es in MPStuBS für jeden vorhandenen Prozessor einen *Idle-Thread*, der immer laufbereit ist. Um diesen *Idle-Thread* wieder zu verlassen, überprüft ein Prozessor nach jedem Interrupt, ob ein neuer *Thread* beim Scheduler angemeldet wurde. Sollte dies der Fall sein, wechselt der Prozessor zu dem neuen *Thread*, ansonsten bleibt er im *Idle-Thread*.

5.4 Synchronisation

Die Synchronisation der Prozessoren soll nach dem in Kapitel 4 vorgestellten Prinzip ablaufen. Auf Kernebene wird nur sequentiell gearbeitet (Kontrollflussebene ohne Verdrängung), d.h. die Datenstrukturen des Kerns müssen nicht zusätzlich geschützt werden, da sich nur ein Prozessor im Kern befinden kann.

Auf Benutzerebene dagegen können Prozesse verdrängt werden (Kontrollflussebene mit Verdrängung), weshalb die Datenstrukturen auf Benutzerebene zusätzlich über gegenseitigen Ausschluss geschützt werden müssen. Dafür werden in MPStuBS Semaphoren verwendet.

5.5 Mono-Prozessor API

Damit alle Anwendungen, die für OOStuBS entwickelt wurden, auch in MPStuBS laufen, soll das *System Call*-API von OOStuBS weiterhin verwendet werden. Die *System Calls* sollen lediglich so erweitert werden, dass die Anwendungen auf allen Prozessoren verteilt laufen, ohne dass der Anwendungsentwickler Konzepte der Verteilung berücksichtigen muss.

Kapitel 6

Entwurf und Implementierung

In diesem Kapitel wird die Implementierung von MPStuBS unter Berücksichtigung der in in Abschnitt 5 festgelegten Anforderungen besprochen.

OOStuBS muss folgendermaßen erweitert werden, um den SMP-Port zu realisieren:

- Erkennung der Systemkonfiguration des SMP-Systems
- Booten der zusätzlichen Prozessoren
- APIC-Programmierung
- Synchronisation der beiden Prozessoren
- Anpassung des Schedulers

6.1 Erkennung der Systemkonfiguration des SMP-Systems

Nachdem das BIOS die Kontrolle an das Betriebssystem übergeben hat, befinden sich alle APs noch im HALT-Zustand. Der BSP führt das Betriebssystem aus, das gegebenenfalls die anderen Prozessoren aktiviert. Dafür werden Informationen über die vorhandene Hardware und deren Konfiguration benötigt. Um dem Betriebssystem diese Informationen zur Verfügung zu stellen, wird vom BIOS eine Datenstruktur angelegt, die *Floating Pointer Structure* [1]. Entweder verweist die Struktur auf die *MP Configuration Table*, die detaillierte Informationen über die Konfiguration des Systems beinhaltet, oder es ist in der Struktur vermerkt, dass eine bekannte Standardkonfiguration verwendet wird. Dann ist die *MP Configuration Table* nicht vorhanden. Abbildung 6.1 zeigt einen Überblick über die MP Datenstrukturen.

Im Folgenden werden die *Floating Pointer Structure* und die *MP Configuration Table*, wie in [1] beschrieben, vorgestellt.

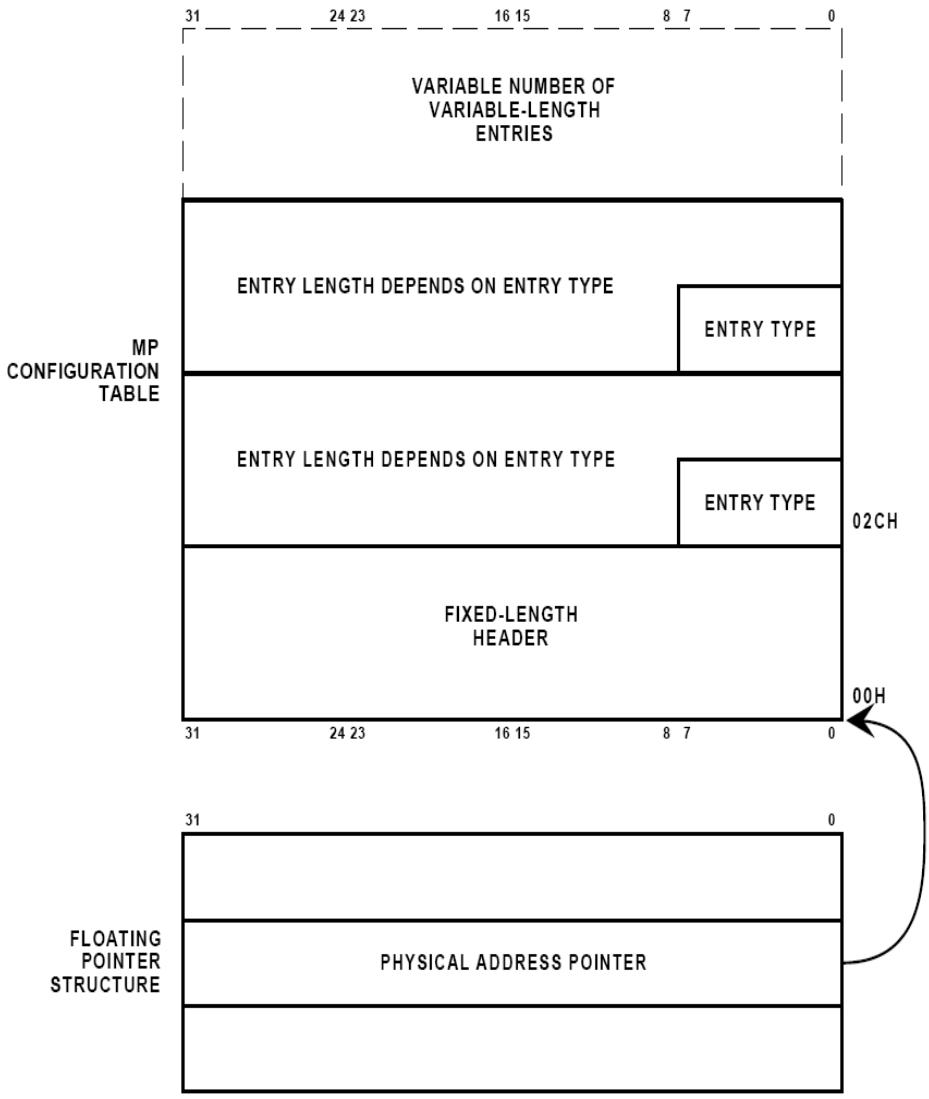


Abbildung 6.1: *MP Floating Pointer Structure* und *MP Configuration Table* ([1] Seite 4-1)

6.1.1 *MP Floating Pointer Structure*

Die *MP Floating Pointer Structure* ist eine 16-Byte große Datenstruktur, die an einer 16-Byte Grenze im Speicher liegt. Sie beinhaltet eine eindeutige Signatur, eine physikalische Adresse, die auf die MP Configuration Table verweist (falls diese vorhanden ist), ihre eigene Größe, die Version der Multi-Prozessor-Spezifikation auf der dieses System beruht, eine Checksumme und zusätzliche Informationen über die Konfiguration des Systems. Abbildung 6.2 zeigt die *MP Floating Pointer Structure*, Tabelle 6.1 beschreibt die einzelnen Felder der Struktur.

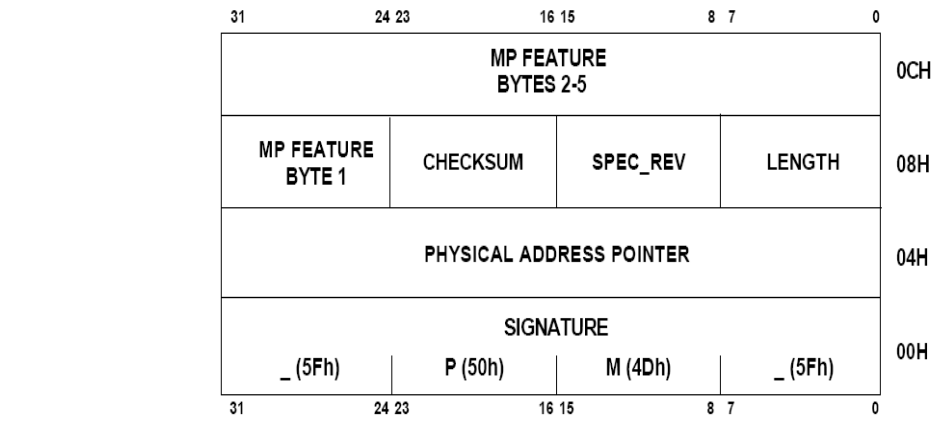


Abbildung 6.2: *MP Floating Pointer Structure* ([1] Seite 4-3)

Feld	**Offset** (bytes:bits)	**Länge** (bits)	**Beschreibung**
Signature	0	32	ASCII String _MP_, der gesucht werden muss, um die Struktur zu finden
Physical Address Pointer	4	32	Anfangsadresse der *MP Configuration Table*, 0 falls sie nicht existiert.
Length	8	8	Länge der MP FPS in 16 Byte Einheiten. 01h da die Struktur 16 Byte groß ist.
Spec_Rev	9	8	Version der verwendeten MP Spezifikation. 01h bedeutet 1.1, 04h 1.4.
Checksum	10	8	Alle Bytes, inklusive der Checksumme und der reservierten Bytes, müssen sich zu 0 aufaddieren.
MP Feature Information Byte 1	11	8	*MP System Configuration Type*. 0, falls die MP Configuration Table vorhanden ist, ansonsten Nummer der verwendeten Standardkonfiguration.
MP Feature Information Byte 2	12:0 12:7	7 1	Reserviert(0). 1, falls das IMCR vorhanden ist und der PIC-Modus verwendet wird, 0 falls *Virtual Wire*-Modus verwendet wird.
MP Feature Information Byte 3-5	13	24	Reserviert(0).

Tabelle 6.1: Felder der MP FPS

Das BIOS plaziert die *MP Floating Pointer Structure* an eine der drei folgenden Stellen im Speicher:

1. Im ersten Kilobyte der *Extended BIOS Data Area* (EBDA)
2. Im letzten Kilobyte des *System Base Memory* (639K - 640K, in einem System mit 640KB *System Base Memory*, 511K - 512K in einem System mit 512KB *System Base Memory*).
3. Im BIOS-ROM Adressraum zwischen F0000h - FFFFFh.

Die Anfangsadresse der EBDA befindet sich an einer zwei Byte Speicherstelle in der *BIOS Data Area* an der Adresse 40:0Eh. Die Größe des *System Base Memory* wird vom BIOS, ebenfalls an eine zwei Byte Speicherstelle in der *BIOS Data Area* an der Adresse 40:13h plaziert.

Um zu überprüfen, ob ein Multi-Prozessor-System vorliegt, muss das Betriebssystem die drei oben genannten Speicherbereiche nach der Signatur „_MP_“ durchsuchen. Wird sie gefunden, kann die Struktur ausgelesen werden und mit den gefundenen Daten das Multi-Prozessor-System gestartet werden, ansonsten handelt es sich um ein Mono-Prozessor-System.

6.1.2 *MP Configuration Table*

Die *MP Configuration Table* ist optional, sollte der *Physical Address Pointer* der *MP Floating Pointer Structure* den Wert 0 besitzen, ist sie nicht vorhanden und es wird eine Standardkonfiguration verwendet (siehe Abschnitt 6.1.3). Ansonsten beinhaltet sie detaillierte Informationen über die Konfiguration des Multi-Prozessor-Systems. Die Tabelle besteht aus einem Header, gefolgt von einer Basissektion und einer erweiterten Sektion, die jeweils Einträge für die einzelnen Hardware-Komponenten und deren Konfiguration enthalten. Abbildung 6.3 zeigt den Header, Tabelle 6.2 beschreibt dessen Felder.

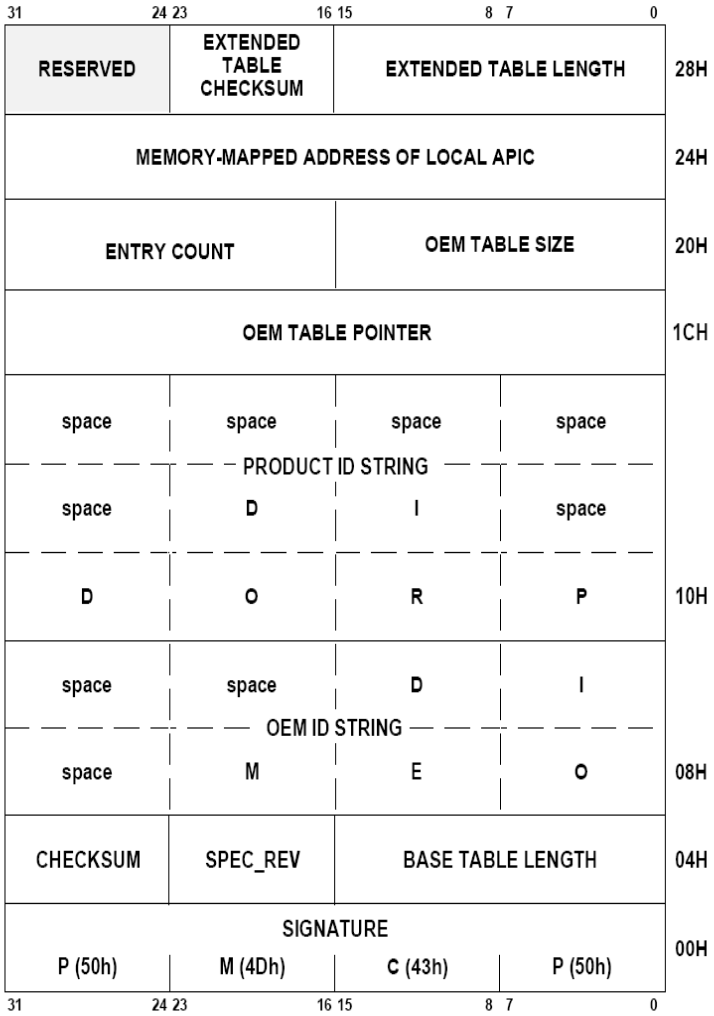


Abbildung 6.3: *MP Configuration Table* Header ([1] Seite 4-5)

Feld	**Offset** (bytes)	**Länge** (bits)	**Beschreibung**
Signature	0	32	ASCII String _PCMP_, der die Struktur eindeutig identifiziert
Base Table Length	4	16	Länge der Basissektion der *Configuration Table*, inklusive des Headers.
Spec_Rev	6	8	Version der verwendeten MP Spezifikation. 01h bedeutet 1.1, 04h 1.4.
Checksum	7	8	Checksumme über die gesamte *Configuration Table*. Alle Bytes, inklusive der Checksumme und der reservierten Bytes, müssen sich zu 0 aufaddieren.
Entry Count	34	16	Anzahl der Einträge in der Basissektion.
Adresse des lokalen APICs	36	32	Die Basisadresse, über die jeder Prozessor seinen lokalen APIC adressiert.

Tabelle 6.2: Felder des *MP Configuration Table* Headers

Basissektion

Die Basissektion beginnt am Ende des *MP Configuration Table* Headers und beinhaltet eine variable Anzahl von Einträgen variabler Länge, die die Hardware des Systems und deren Konfiguration beschreiben. Alle Einträge der Basissektion sind abwärtskompatibel mit früheren Versionen der Intel MP Specification [1]. Es gibt mehrere Typen von Einträgen, die jeweils eine feste Länge haben. Die Größe der Sektion hängt also von der Anzahl und dem Typ der Einträge ab. Das Betriebssystem kann bis zu ENTRY COUNT Einträge auslesen. Die Einträge sind in aufsteigender Reihenfolge nach ihrem Typ sortiert. Tabelle 6.3 zeigt alle verschiedenen Typen.

Eintrag	**Typ**	**Länge** (in bits)	**Beschreibung**
Prozessor	0	20	Ein Eintrag pro Prozessor
Bus	1	8	Ein Eintrag pro Bus
I/O APIC	2	8	Ein Eintrag pro I/O APIC
I/O Interrupt Zuweisung	3	8	Ein Eintrag pro Bus Interrupt Quelle
lokale Interrupt Zuweisung	4	8	Ein Eintrag pro System Interrupt Quelle

Tabelle 6.3: Einträge der Basis Sektion

Da in MPStuBS nur der Prozessoreintrag ausgelesen wird, wird im Folgenden nur dieser Eintrag genauer beschrieben. Eine Beschreibung der anderen Einträge findet man in der Intel MP Specification [1].

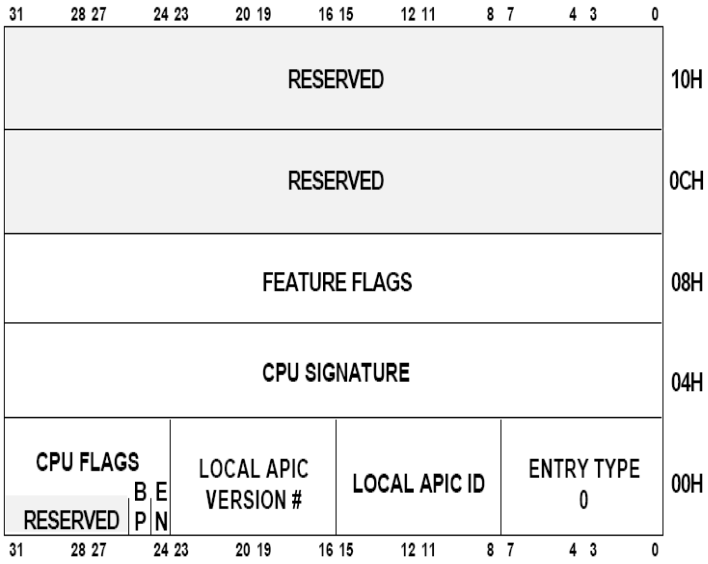


Abbildung 6.4: Prozessoreintrag in der Basissektion ([1] Seite 4-7)

Feld	**Offset** (bytes:bits)	**Länge** (bits)	**Beschreibung**
Entry Type	0	8	Der Wert 0 identifiziert einen Prozessor-Eintrag.
Local APIC ID	1	8	APIC ID des diesem Prozessor zugehörigen lokalen APICs.
Local APIC Version #	2	8	Bits 0-7 des Versionsregisters des lokalen APICs.
CPU Flags EN	3:0	1	Falls 0, ist der Prozessor unbrauchbar.
CPU Flags BP	3:1	1	1, falls der Prozessor der BSP ist.

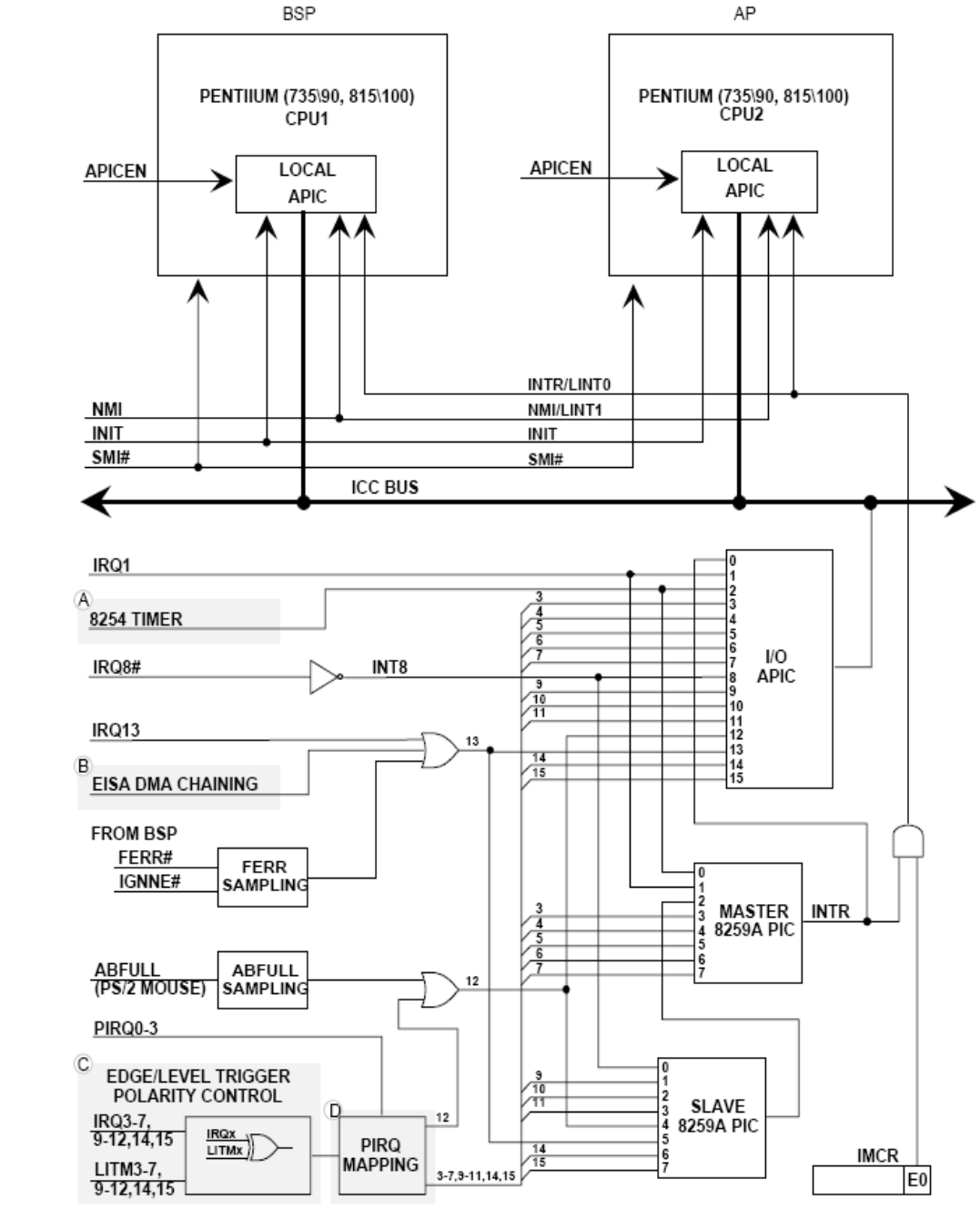
Tabelle 6.4: Felder eines Prozessor Eintrags

Erweiterte Sektion

Da die erweiterte Sektion in MPStuBS nicht benötigt wurde, wird hier nicht weiter darauf eingegangen. Genaueres findet man in der Intel MP Specification [1].

6.1.3 Standardkonfigurationen

Sollte keine *MP Configuration Table* vorhanden sein, liegt eine Standardkonfiguration vor. Standardkonfigurationen gibt es nur in Systemen mit zwei Prozessoren. Abbildung 6.5 zeigt eine Standardkonfiguration in der ein integrierter lokaler APIC verwendet wird. Detaillierte Beschreibungen aller Standardkonfigurationen findet man in der Intel MP Specification [1].



SHADED AREAS:
A,B: MAY NOT BE EXTERNALIZED WITH SOME EISA CHIPSETS
B,C: EISA BUS SPECIFIC
D: PCI BUS SPECIFIC

Abbildung 6.5: Default Konfiguration mit integriertem lokalen APIC

6.1.4 MPStuBS-Code-Referenzen

- **int smp_detect(void)**: Durchsucht die drei Speicherbereiche und ruft dafür bis zu dreimal die Methode smp_search_config auf.
- **static int smp_search_config(unsigned long base, unsigned long length)**: Durchsucht den Speicherbereich beginnend mit der Adresse base bis zu der Adresse base+length nach dem String „_MP_“. Sollte er gefunden werden, wird die MP Floating Pointer Structure ausgelesen und für den Fall, dass eine MP Configuration Table existiert, die Methode smp_read_mpct aufgerufen.
- **static int smp_read_mpct(struct mp_config_table_header mpcth)**: Liest die MP Configuration Table aus, sichert die nötigen Daten, und gibt die Anzahl der Prozessoren zurück.

6.2 Booten der zusätzlichen Prozessoren

Nachdem das Multi-Prozessor-System erfolgreich erkannt wurde, und die *MP Configuration Table* vollständig ausgelesen oder eine Standardkonfiguration erkannt wurde, ist die Konfiguration des Systems und damit die Anzahl und die IDs der APs bekannt. Die APs befinden sich im HALT-Zustand, Interrupts sind abgeschaltet und die APs reagieren nur auf die speziellen Inter-Prozessor-Interrupts (IPIs) INIT und STARTUP [1]. Um einen solchen IPI an einen bestimmten AP zu schicken, benötigt der BSP lediglich die inzwischen bekannte ID des APs.

Je nach Version des lokalen APICs wird entweder der INIT oder der STARTUP IPI verwendet, um einen Prozessor aufzuwecken. Der folgende universelle Algorithmus (in Pseudocode) weckt einen Prozessor unabhängig von dem vorliegenden lokalen APIC auf, da, falls nötig, sowohl INIT als auch STARTUP IPIs geschickt werden. Bevor er ausgeführt wird, muss der BSP den *BIOS Shutdown Code* auf 0Ah setzen, und den *Warm Reset Vector* auf den *Startup Code* für den AP zeigen lassen (Genaueres dazu im Abschnitt 6.2.1).

```
BSP schickt AP INIT IPI
BSP verzögert (10 millisec)
IF(APIC Version ist nicht 82489DX)
 {
    BSP schickt AP STARTUP IPI
    BSP verzögert (200microsec)
    BSP schickt AP STARTUP IPI
    BSP verzögert (200microsec)
 }
BSP prüft Synchronisation mit dem AP
```

Das Betriebssystem muss überprüfen, ob der lokale APIC die IPIs erfolgreich verschickt hat. Dafür muss das *Delivery Status Bit* im *Interrupt Command Register*

(ICR) überprüft werden, das vom lokalen APIC bei erfolgreicher Versendung des Interrupts zurückgesetzt wird (mehr zum Versenden von IPIs im Abschnitt 6.3.1). 20 Mikrosekunden reichen dem APIC, um einen Interrupt zu verschicken. Sollte nach dieser Zeitspanne das *Delivery Status Bit* nicht zurückgesetzt worden sein, wird das Booten dieses Prozessors entweder abgebrochen oder ein weiterer Interrupt versendet.

Des Weiteren ist das Betriebssystem dafür verantwortlich, zu überprüfen, ob die Interrupts beim adressierten APIC angekommen sind, und der entsprechende AP tatsächlich aufgeweckt wurde. Dafür kann z.B. ein *Status Flag* angelegt werden, das der AP, nachdem er aufgeweckt wurde, setzt, um zu signalisieren, dass er läuft. Sollte dieses *Flag* nach einer bestimmten Zeitspanne nicht gesetzt sein, behandelt der BSP den entsprechenden AP als nicht vorhanden oder nicht funktionsfähig.

Im Folgenden wird gezeigt, wie die Inter-Prozessor-Interrupts INIT und STARTUP verwendet werden, um einen Prozessor aufzuwecken.

6.2.1 INIT IPI

In Systemen, die den 82489DX APIC als lokalen APIC verwenden, werden die APs mit einem INIT IPI aufgeweckt [1]. Der INIT IPI ist ein *Level Triggered* Interrupt. Er muss zunächst angelegt werden, und dann wieder zurückgenommen werden.

Wenn ein lokaler APIC ein INIT IPI empfängt, löst er bei seinem Prozessor ein INIT aus. Das führt dazu, dass der Prozessor auf einen initialen Zustand zurückgesetzt wird. Ausnahmen sind die *Caches*, die *Floating Point Unit* und die *Write Buffer*, die nicht zurückgesetzt werden. Danach beginnt der Prozessor mit der Ausführung von Befehlen an einer festgelegten Adresse, die er im *Reset Vector* findet. Um also einen Prozessor an einer bestimmten Adresse aufzuwecken, muss der INIT IPI als Teil eines *Warm Reset* verwendet werden.

Der *Warm Reset* ist eine Eigenschaft, die es ermöglicht, den Prozessor zu initialisieren, ohne dabei die komplette BIOS Initialisierungs Prozedur (POST) zu durchlaufen. Die beiden folgenden Schritte müssen für einen *Warm Reset* durchgeführt werden [1]:

1. Der *Shutdown Code* (Adresse 0Fh im CMOS RAM), der während dem POST gelesen wird, um den Grund für den Reset herauszufinden, muss auf 0Ah gesetzt werden. 0Ah repräsentiert einen *Warm Reset*.

2. Sollte der POST einen *Shutdown Code* von 0Ah finden, führt er einen indirekten Sprung über den *Warm Reset Vector* (Adresse 40:47h im System RAM) aus. Dieser Vektor muss auf die gewünschte Zieladresse des Prozessors gesetzt werden.

6.2.2 STARTUP IPI

In Systemen, die lokale APICs ab der Version 1.x verwenden, wird der STARTUP IPI verwendet, um einen Prozessor aufzuwecken [1]. Der STARTUP IPI ist ein *Edge Triggered* Interrupt.

Ein STARTUP IPI führt dazu, dass der adressierte Prozessor im *Real Mode* aufgeweckt wird und an der Adresse 000VV000h mit der Ausführung von Befehlen beginnt. VV ist ein 8 Bit Vektor, der Teil des STARTUP IPIs ist. Die resultierende Startadresse ist eine 4 Kilobyte Grenze im ersten Megabyte des Adressraums, die Vektoren A0h-BFh sind jedoch reserviert und dürfen nicht verwendet werden. Ein STARTUP IPI verändert außer dem *Instruction Pointer* nichts am Zustand des Prozessors. Auch wenn sich der Prozessor im HALT-Zustand direkt nach einem Reset oder einem INIT befindet, nimmt er den STARTUP IPI an, und verlässt diesen Zustand, um an der Adresse 000VV000h mit der Ausführung von Befehlen zu beginnen.

Einem 82489DX APIC sollten keine STARTUP IPIs geschickt werden, da er diese ignoriert.

6.2.3 Prozessor Start

Unabhängig davon, ob INIT oder STARTUP IPIs verwendet werden, ist das Betriebssystem in der Lage, einen AP aufzuwecken, indem es den *Startup Code* für den Prozessor an eine bestimmte Adresse schreibt, und den Prozessor dann über einen IPI dazu veranlasst, an dieser Adresse mit der Ausführung von Befehlen zu beginnen.

Da der Prozessor im funktional eingeschränkten 16 Bit *Real Mode* aufwacht, muss in den 32 Bit *Protected Mode* gewechselt werden, bevor in den Kern gesprungen werden kann. Dafür sind folgende Schritte nötig:

- im 16 Bit Code
  - Sowohl maskierbare als auch der *Non Maskable Interrupt* sollten verboten werden.
  - Ein GDT für den *Protected Mode* muss angelegt werden, und das *Global Descriptor Table Register* (GDTR) mit dessen Basisadresse geladen werden. Der Einfachheit halber wird für MPStuBS ein GDT angelegt, der nur zwei Deskriptoren beinhaltet, einen für das Code-Segment und einen für das Daten-Segment. Beide beginnen bei der Adresse 0 und sind vier Gigabyte groß, sie entsprechen also dem kompletten Adressraum. Das Code-Segment darf gelesen und ausgeführt, das Daten-Segment gelesen und geschrieben werden.
  - Die Umschaltung in den *Protected Mode* muss erfolgen. Dafür muss das niederwertigste Bit des Steuerregisters CR0 gesetzt werden.

  - Ein *Far Jump* in den *Protected Mode* Code (32 Bit) des Systems muss erfolgen.

- im 32 Bit Code
  - Die Segmentregister müssen mit den richtigen Segmentselektoren geladen werden (CS wird beim *Far Jump* korrekt beschrieben, DS, ES, FS, GS und SS müssen alle mit dem Selektor des Daten-Segments beschrieben werden).
  - Der *Stack* muss anlegt, und der *Stackpointer* (ESP) gesetzt werden.
  - Als letztes kann eine Routine des Kerns aufgerufen werden.

6.2.4 MPStuBS-Code-Referenzen

- **void smp_init(void)**: Bootet die APs über die Methode smp_boot_cpus und initialisiert dann den lokalen, und den I/O-APIC.
- **void smp_boot_cpus(void)**: Bootet jeden vorhandenen Prozessor über die Methode do_boot_cpu.
- **int do_boot_cpu(int id)**: Führt den universellen Algorithmus zum Aktivieren eines Prozessors aus.
- **void smp_delay(int time)**: Wartet time*10 Millisekunden.
- **setup2nd.asm**: Wechselt vom *Real* in den *Protected Mode*.
- **startup2nd.asm**: Initialisiert die Segmentregister und springt in den Kern.

6.3 APIC-Programmierung

Wenn der Prozessorstart korrekt abgelaufen ist, sind alle Prozessoren aktiv und in der Lage, Betriebssystem-Code auszuführen. Da in MPStuBS alle Prozessoren gleichberechtigt sind, sollen die Interrupts unter ihnen gleichmäßig verteilt werden.

Wenn ein Prozessor einen Interrupt empfängt, verwendet er dessen Vektornummer als Index in die *Interrupt Descriptor Table* (IDT), die für jeden Interrupt einen Verweis auf die entsprechende Interrupt-Behandlungsroutine beinhaltet [4]. Um Interrupts bearbeiten zu können, muss die IDT angelegt sein, und das *Interrupt Descriptor Table Register* (IDTR) auf die Anfangsadresse der IDT gesetzt werden. Dann wird bei jedem Interrupt der entsprechende Eintrag aus der IDT gelesen und die entsprechende Interrupt Behandlungs Routine ausgeführt. In OOStuBS wird diese Tabelle im *Startup Code* des Prozessors angelegt. Da alle Prozessoren Interrupts gleich behandeln sollen, werden in MPStuBS die IDTRs aller Prozessoren mit der gleichen Adresse (der Anfangsadresse des im *Startup Code* des BSP erzeugten IDT) geladen. Das Laden des IDTRs wird bei MPStuBS für jeden Prozessor im *Startup Code* durchgeführt.

Wie in Abschnitt 3.1.2 beschrieben, wird das System entweder im PIC- oder im *Virtual Wire*-Modus gestartet, d.h. alle Interrupts werden zum BSP geschickt. Um die Interrupts auf alle vorhandenen Prozessoren zu verteilen, müssen die APICs programmiert werden.

6.3.1 Lokaler APIC

Jeder Prozessor verfügt über einen lokalen APIC, der lokale, externe (vom I/O-APIC) und Inter-Prozessor-Interrupts empfängt, und in der Lage ist, Inter-Prozessor-Interrupts an andere lokale APICs zu schicken [4]. Die Programmierung des APICs erfolgt über seinen Registersatz, der beginnend mit der Adresse FEE00000h in eine vier Kilobyte Seite des Speichers gemappt ist. Die Register sind 32, 64 oder 256 Bit breit, liegen an 128 Bit Grenzen im Speicher und können vom Betriebssystem direkt adressiert werden. Die in MPStuBS verwendeten werden im Folgenden vorgestellt.

ID Register

Die APIC ID steht im *ID Register* und wird jedem lokalen APIC beim Systemstart von der Hardware zugewiesen. Sie wird auch als Prozessor ID verwendet. Da der CPUID Befehl (über den, ebenso wie über das *ID Register*, die ID des Prozessors bestimmt werden kann) immer die initiale, von der Hardware zugewiesene ID zurückgibt, sollte das Betriebssystem den Wert des *ID Register* nicht verändern. Abbildung 6.6 zeigt das *ID Register*.

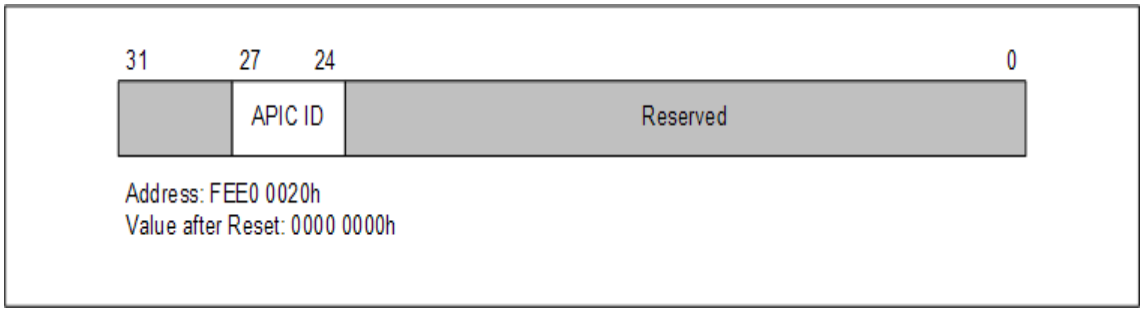


Abbildung 6.6: *ID Register* (nach [4] Seite 8-13)

Spurious Interrupt Vector Register (SVR)

Das *Spurious Interrupt Vector Register* wird unter anderem verwendet, um einen lokalen APIC zu aktivieren oder zu deaktivieren. Dafür muss das entsprechende Bit gesetzt oder zurückgesetzt werden. Abbildung 6.7 zeigt das SVR.

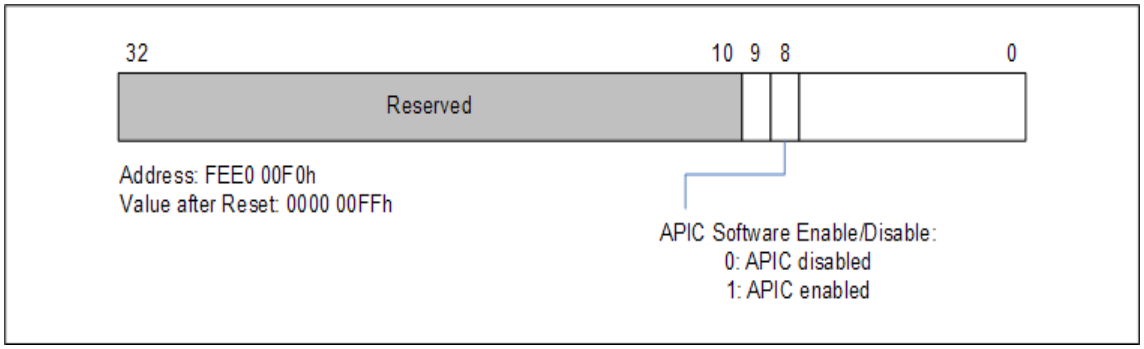


Abbildung 6.7: *Spurious Interrupt Vector Register* (nach [4] Seite 8-45)

End of Interrupt (EOI) *Register*

Mit Ausnahme von NMI, SMI, INIT, ExtINT und STARTUP muss für alle Interrupts, die in MPStuBS verwendet werden, signalisiert werden, dass sie akzeptiert und behandelt wurden. Dies erfolgt, indem die Interrupt-Behandlungsroutine den Wert 0 in das EOI Register schreibt, sobald weitere Interrupts empfangen werden können. Das EOI Register liegt an der Adresse FEE000B0h.

Task Priority Register (TPR)

Das *Task Priority Register* beinhaltet die Priorität des laufenden Prozesses. Sollen Prozesse mit unterschiedlichen Prioritäten bearbeitet werden, muss das TPR bei jedem Prozesswechsel neu beschrieben werden (in MPStuBS ist das nicht nötig, alle Prozesse haben die Priorität 0). Interrupts werden nur bearbeitet, wenn sie eine höhere Priorität haben als der laufende Prozess. Die niedrigste Priorität ist 0, die höchste 15. Ist das TPR auf 0 gesetzt, werden alle Interrupts behandelt, steht es auf 15, werden nur NMI, SMI, INIT, ExtINT und STARTUP bearbeitet. Abbildung 6.8 zeigt das *Task Priority Register*.

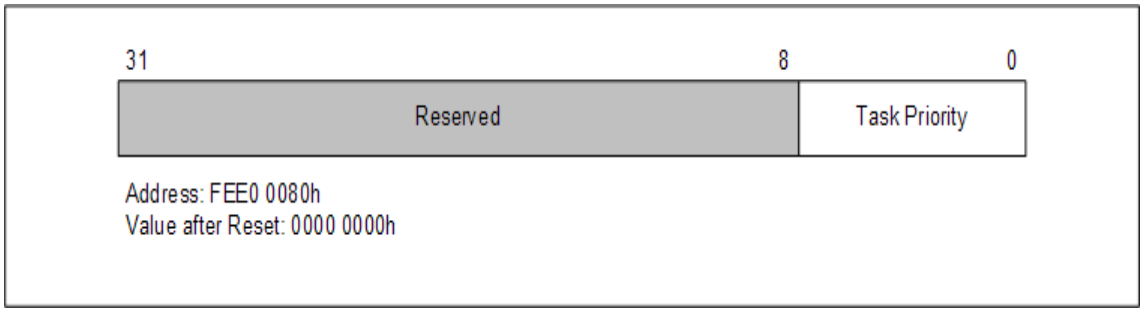


Abbildung 6.8: *Task Priority Register* (nach [4] Seite 8-39)

Die Priorität der Interrupts errechnet sich aus der Vektornummer (die vom I/O-APIC gesetzt werden kann, mehr dazu in Abschnitt 6.3.2) über folgende Formel:

Priorität = Vektornummer/16

Das Ergebnis wird abgerundet, die niedrigste Priorität ist 1, die höchste 15. Nachdem die Vektornummern 0 bis 31 reserviert sind, liegt die Priorität für benutzerdefinierte Interrupts zwischen 2 und 15.

Interrupt Command Register (ICR)

Das *Interrupt Command Register* wird verwendet, um Inter-Prozessor-Interrupts an andere lokale APICs zu schicken. Es ist ein 64 Bit Register, das über zwei Adressen beschrieben werden kann. Um einen Interrupt zu schicken, müssen zunächst die adressierten Prozessoren in die oberen 32 Bit des Registers geschrieben werden, das Schreiben der unteren 32 Bit löst dann das Verschicken des Interrupts aus. Abbildung 6.9 zeigt das *Interrupt Command Register*, Tabelle 6.5 beschreibt die in MPStuBS relevanten Felder.

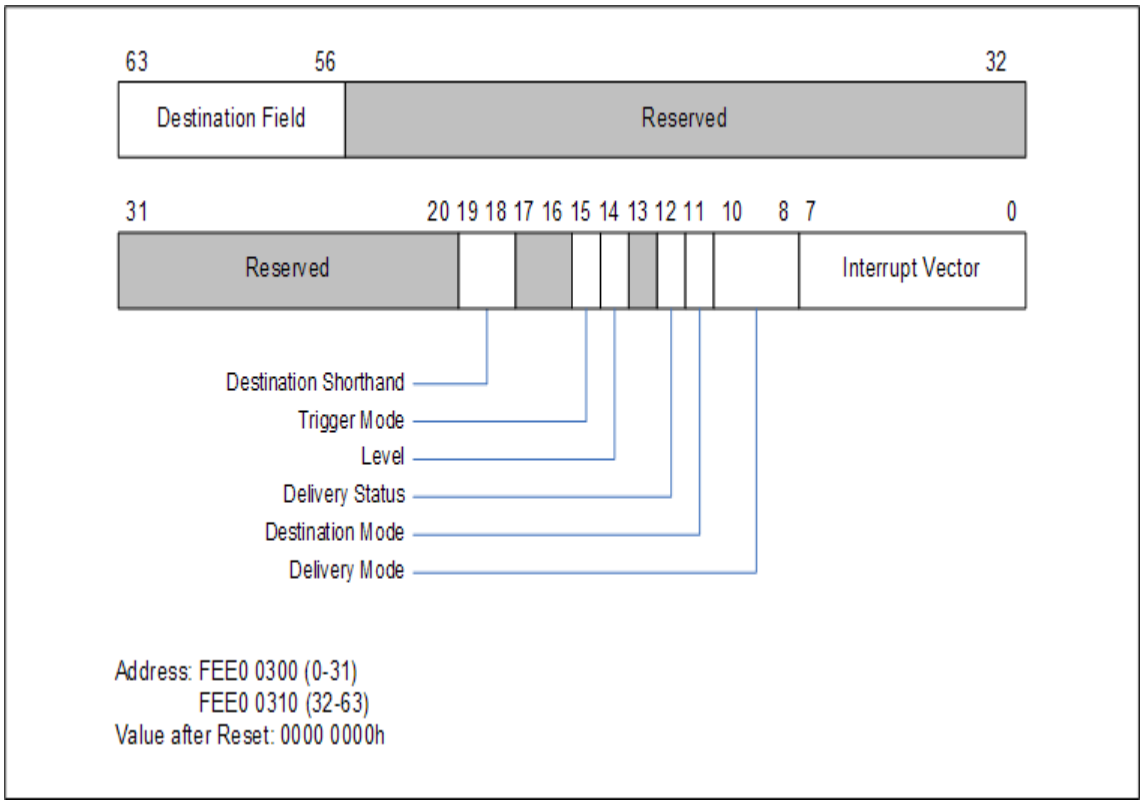


Abbildung 6.9: *Interrupt Command Register* (nach [4] Seite 8-24)

6.3.2 I/O-APIC

Jedes Multi-Prozessor-System verfügt über mindestens einen I/O-APIC, der es ermöglicht, Interrupts sowohl statisch als auch dynamisch auf alle vorhandenen Prozessoren zu verteilen [6]. Dafür verfügt der I/O-APIC über eine *Interrupt Redirection Table*, die für jeden möglichen Interrupt einen Eintrag beinhaltet, der beschreibt, ob und an welchen Prozessor der entsprechende Interrupt geschickt werden soll.

Feld	Beschreibung
Vector	Die Vektornummer des zu verschickenden Interrupts.
Delivery Mode	*Fixed*: Schickt den Interrupt an die Zielprozessoren. *Lowest Priority*: Schickt den Interrupt an den Zielprozessor, der mit der geringsten Priorität arbeitet. INIT: Schickt einen INIT IPI an die Zielprozessoren, das Vektor-Feld muss auf 0 gesetzt sein. STARTUP: Schickt einen STARTUP IPI an die Zielprozessoren.
Destination Mode	Legt fest, ob der *Physical*(0) oder der *Logical*(1) *Destination Mode* verwendet wird (siehe Abschnitt 6.3.3)
Delivery Status *(read only)*	*Idle*(0): Es herrscht keine IPI Aktivität, oder der letzte IPI, der von diesem APIC verschickt wurde, wurde bereits vom adressierten APIC akzeptiert. *Send Pending*(1): Der letzte von diesem APIC verschickte IPI wurde noch nicht vom adressierten APIC akzeptiert.
Level	Legt für den INIT IPI (*level triggered*) fest, ob er angelegt(1) oder zurückgenommen(0) wird.
Trigger Mode	Legt den *Trigger Mode* für einen Interrupt fest. *edge*(0) oder *level*(1).
Destination Shorthand	Legt fest ob eine Abkürzung für die Ziel-Prozessoren verwendet wird. *No Shorthand*(00), *Self*(01), *All Including Self*(10) oder *All Excluding Self*(11)
Destination Field	Legt über eine MDA oder ein physikalische Adresse fest, welche Prozessoren adressiert werden, falls kein *Destination Shorthand* verwendet wird. (siehe Abschnitt 6.3.3)

Tabelle 6.5: Felder des ICR

Die Programmierung des I/O-APICs erfolgt über zwei seiner Register(*I/O Register Select Register* und *I/O Window Register*), die verwendet werden, um die anderen Register indirekt zu adressieren. Diese beiden Register sind in den Speicher gemappt und können vom Betriebssystem direkt adressiert werden. Im Folgenden werden die für MPStuBS relevanten Register vorgestellt.

I/O Register Select Register

Über das *I/O Register Select Register* wird festgelegt, welches Register gelesen oder geschrieben werden soll. Dafür muss der für jedes Register eindeutige Offset ins *I/O Register Select Register* geschrieben werden. Abbildung 6.10 zeigt das *I/O Register Select Register*.

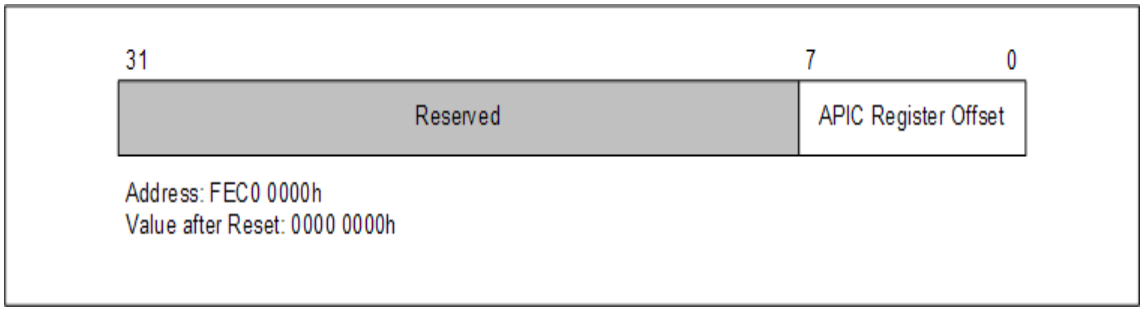


Abbildung 6.10: *I/O Register Select Register*

I/O Window Register

Das *I/O Window Register* wird verwendet, um das über das *I/O Register Select Register* selektierte Register zu lesen oder zu schreiben. Abbildung 6.11 zeigt das *I/O Window Register*.

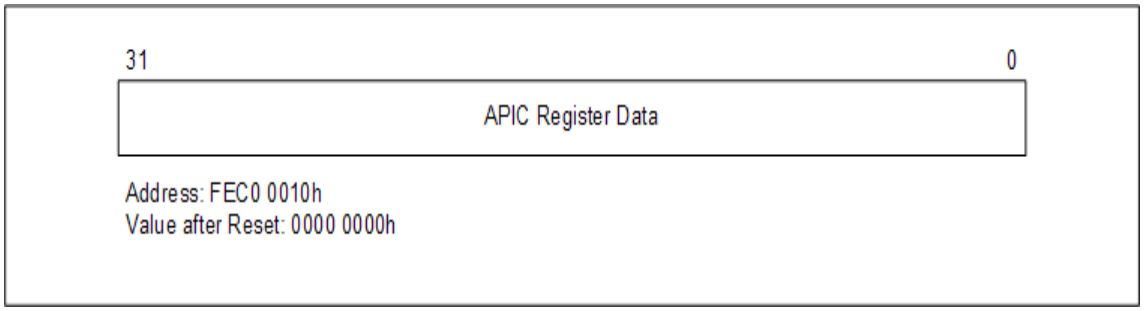


Abbildung 6.11: *I/O Window Register*

ID Register

Das *ID Register* beinhaltet die 4 Bit ID des I/O-APICs. Die ID wird als Name des APICs verwendet und muss mit dem korrekten Wert (kann aus der *MP Configuration Table* gelesen werden) beschrieben werden, bevor der APIC zum Verschicken von Nachrichten verwendet wird. Abbildung 6.12 zeigt das *ID Register*.

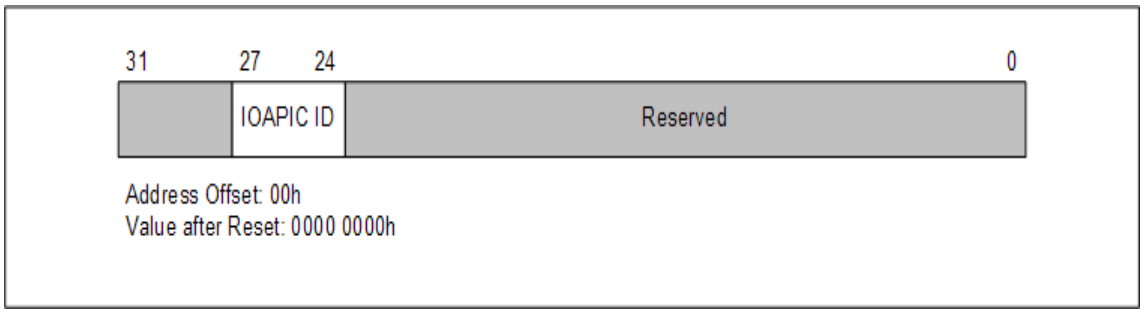


Abbildung 6.12: *ID Register*

I/O Redirection Table Register

Es gibt 24 jeweils 64 Bit breite *Redirection Table Register*. Jedes dieser Register ist einem der Interrupt *Input Pins* des I/O-APICs zugeordnet und beschreibt die Behandlung dieses Interrupts. Über die *Redirection Table Register* kann das Betriebssystem jedem Interrupt unter anderem eine Vektornummer (und damit eine Priorität) zuordnen und den oder die Zielprozessoren sowie den Verteilungsmodus (Destination Mode) festlegen. Die Informationen in den *Redirection Table* Registern verwendet der I/O-APIC, um Interrupt Nachrichten zu erzeugen, die er über den APIC Bus an einen oder mehrere lokale APICs verschickt. Abbildung 6.13 zeigt ein *Redirection Table Register*, Tabelle 6.6 beschreibt dessen Felder.

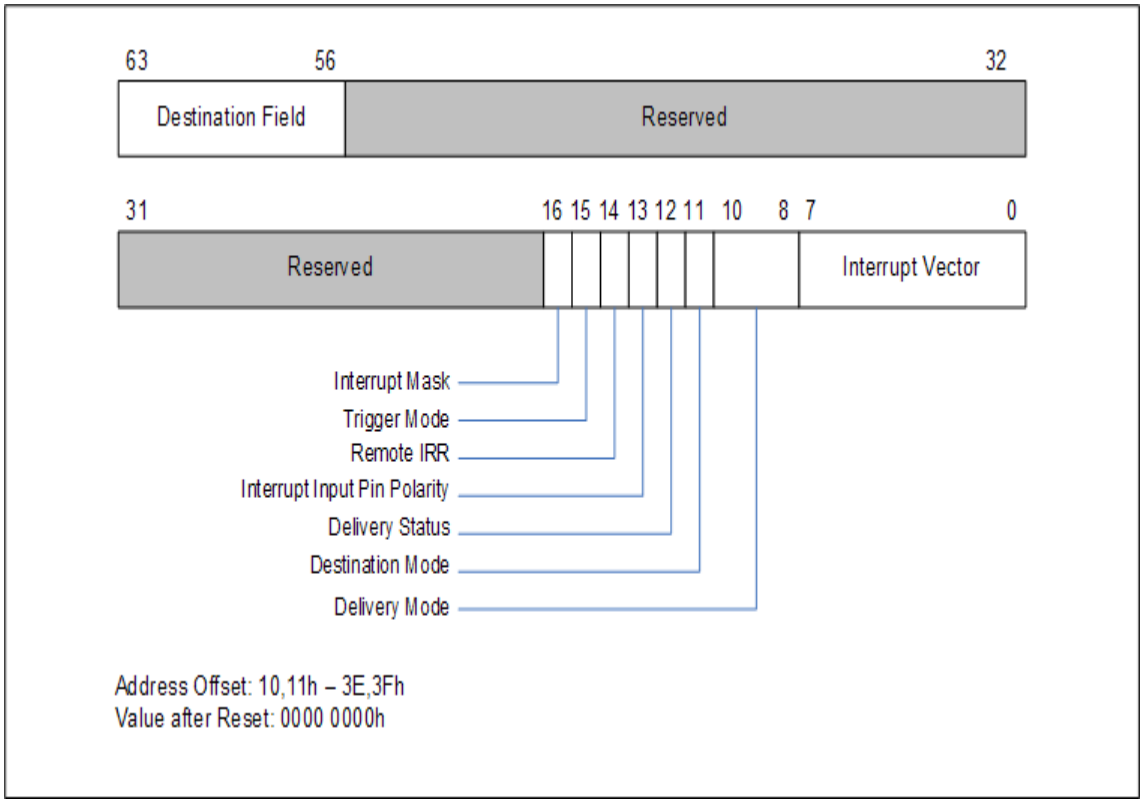


Abbildung 6.13: *I/O Redirection Table Register*

Feld	Beschreibung
Vector	Die Vektornummer des zu verschickenden Interrupts.
Delivery Mode	*Fixed*: Schickt den Interrupt an die Zielprozessoren. *Lowest Priority*: Schickt den Interrupt an den Zielprozessor, der mit der geringsten Priorität arbeitet.
Destination Mode	Legt fest, ob der *Physical*(0) oder der *Logical*(1) *Destination Mode* verwendet wird (siehe Abschnitt 6.3.3)
Delivery Status (*read only*)	*Idle*(0): Es herrscht keine Interrupt Aktivität, oder der letzte Interrupt, der von diesem APIC verschickt wurde, wurde bereits vom adressierten APIC akzeptiert. *Send Pending*(1): Der letzte von diesem APIC verschickte Interrupt wurde noch nicht vom adressierten APIC akzeptiert.
Interrupt Input Pin Polarity	Beschreibt die Polarität des Interrupts. *High active*(0) oder *Low active*(1).
Remote IRR (*read only*)	Wird bei *level triggered* Interrupts verwendet. 1, wenn der adressierte APIC den Interrupt akzeptiert. 0, wenn ein EOI empfangen wurde.
Trigger Mode	Legt den *Trigger Mode* für einen Interrupt fest. *edge*(0) oder *level*(1).
Interrupt Mask	1, wenn der Interrupt maskiert ist, er wird dann ignoriert. 0, wenn er nicht maskiert ist.
Destination Field	Legt über eine MDA oder ein physikalische Adresse fest, welche Prozessoren adressiert werden, falls kein *Destination Shorthand* verwendet wird. (siehe Abschnitt 6.3.3)

Tabelle 6.6: Felder des ICR

6.3.3 APIC *Destination Mode*

Ein APIC muss beim Verschicken eines Interrupts festlegen, ob er den Interrupt an einen, alle oder eine Gruppe von Prozessoren schicken möchte. Dafür verwendet er folgende Register [4]:

- *Interrupt Command Register*
- *I/O Redirection Table Registers*
- *Logical Destination Register*(LDR)
- *Destination Format Register*(DFR)

Wie diese Register verwendet werden, um die Zielprozessoren festzulegen, hängt vom *Delivery Mode* und vom *Destination Mode* ab. In MPStuBS werden der *Physi-*

cal Destination-, der *Logical Destination*- und der *Lowest Priority Delivery*-Modus verwendet.

Physical Destination Mode

Im *Physical Destination Mode* kann nur ein Prozessor adressiert werden. Für die Adressierung wird die ID des lokalen APICs (siehe Abschnitt 6.3.1) des Zielprozessors verwendet. Der den Interrupt versendende APIC schreibt diese ID in das Ziel-Feld (*Destination Field*) des *Interrupt Command Register* oder des *I/O Redirection Table Registers*. Die anderen oben genannten Register werden in diesem Modus nicht benötigt.

Logical Destination Mode

Im *Logical Destination Mode* können beliebig viele Prozessoren adressiert werden. Für die Adressierung wird eine 8 Bit *Message Destination Address* (MDA) verwendet, die in das Ziel-Feld (*Destination Field*) des *Interrupt Command Register* oder des *I/O Redirection Table* Registers geschrieben wird. Wenn ein lokaler APIC einen Interrupt, der in diesem Modus verschickt wurde, empfängt, vergleicht er die MDA der Nachricht mit den Werten im LDR und im DFR um festzustellen, ob er diesen Interrupt bearbeiten soll. Abbildung 6.14 zeigt das LDR. Die 8 Bit *logical* APIC ID wird mit der MDA verglichen.

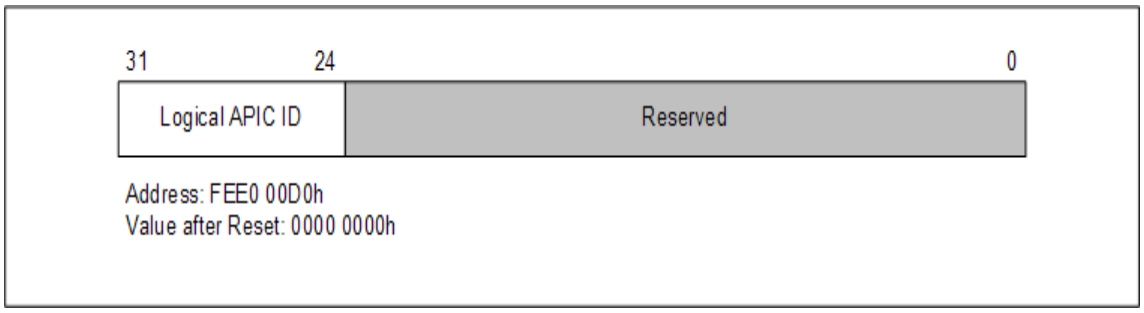


Abbildung 6.14: *Logical Destination Register* (nach [4] Seite 8-31)

Abbildung 6.15 zeigt das DFR. Das 4 Bit Modell-Feld legt fest, ob das *Flat*- oder das *Cluster*-Modell verwendet wird, um die MDA zu interpretieren.

Beim *Flat*-Modell ist es möglich, bis zu acht Prozessoren eine eigene ID zuzuweisen, indem für jeden lokalen APIC ein anderes der acht Bit der *logical* APIC ID im LDR gesetzt wird. Eine Gruppe von Prozessoren kann adressiert werden, indem ein oder mehrere Bits in der MDA gesetzt werden. Jeder lokale APIC führt einen Bit-weisen UND-Vergleich der MDA und der *logical* APIC ID durch. Liefert einer der acht Vergleiche eine 1, wird der empfangene Interrupt bearbeitet.

Da in MPStuBS nur das *Flat*-Modell verwendet wird, wird hier auf das *Cluster*-Modell nicht genauer eingegangen. Eine detaillierte Beschreibung des *Cluster*-Modells findet man in [4].

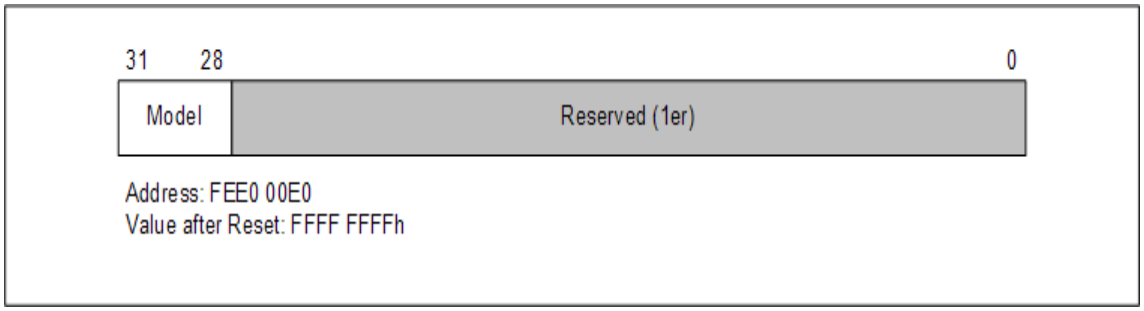


Abbildung 6.15: *Destination Format Register* (nach [4] Seite 8-31)

Lowest Priority Delivery Mode

Der *Lowest Priority Delivery Mode* ist ein Modus, in dem der *Logical Destination Mode* verwendet wird, um eine Gruppe von Prozessoren zu adressieren, von denen aber nur der Prozessor den Interrupt bearbeitet, der im Moment des Empfangens mit der niedrigsten *Task*-Priorität arbeitet. Sollten mehrere Prozessoren die niedrigste Priorität haben, werden die Interrupts über ein *Round Robin*-Verfahren unter ihnen verteilt [4].

6.3.4 MPStuBS-Code-Referenzen

- Die Klasse LAPIC repräsentiert den lokalen APIC und verfügt über alle nötigen Methoden, um ihn zu konfigurieren.
- Die Klasse IO_APIC repräsentiert den I/O-APIC und verfügt über alle nötigen Methoden, um ihn zu konfigurieren.

6.4 Synchronisation der Prozessoren

In einer *Tightly Coupled*, *Shared Memory*, *Symmetric* Multi-Prozessor-Architektur teilen sich alle Prozessoren den Speicher und sind somit in der Lage, mit den gleichen Daten zu arbeiten. Das kann zu Inkonsistenzen führen, falls mehrere Prozessoren gleichzeitig auf die gleichen Daten zugreifen und diese verändern. In MPStuBS treten solche Probleme z.B. beim Zugriff auf die globale *Readylist* (auf Kernebene) oder bei der Bildschirmausgabe (auf Benutzerebene) auf. Um die Konsistenz dieser Daten zu gewährleisten, müssen Zugriffe darauf synchronisiert werden.

Die Kernebene wird als Kontrollflussebene ohne Verdrängung implementiert, d.h. es muss lediglich der Eintritt in diese Ebene synchronisiert werden, um die Daten des Kerns zu schützen. In OOStuBS wird das Betreten des Kerns über eine Schlossvariable synchronisiert. Da in MPStuBS mehrere Prozessoren gleichzeitig auf diese Variable zugreifen können, muss sie durch ein *Spinlock* ersetzt werden. Dieses *Spinlock* wird also bei jedem *System Call* gesetzt und gewährleistet somit, dass sich immer nur ein Prozessor im Kern befindet.

Die Benutzerebene wird als Kontrollflussebene mit Verdrängung implementiert, d.h. es wird gegenseitiger Ausschluss benötigt, um die Daten auf Benutzerebene vor gleichzeitigem Zugriff durch mehrere Prozessoren zu schützen. Dafür werden in MPStuBS Semaphoren verwendet.

6.4.1 *Spinlocks*

Beim Schließen eines *Spinlocks* muss in einer while-Schleife der Wert des Schlosses abgefragt werden. Solange es geschlossen ist, bleibt der Prozess in der Schleife, er wartet aktiv. Sobald es frei ist, wird es geschlossen und die Schleife verlassen. Das Überprüfen und gegebenenfalls Schließen des Spinlocks muss atomar erfolgen und wird in MPStuBS vom GCC Built-In __sync_lock_test_and_set übernommen. Das Öffnen des Schlosses muss nicht atomar erfolgen.

6.4.2 Semaphoren

Die Implementierung von Semaphoren wurde direkt aus OOStuBS übernommen. Da der Zugriff auf Semaphoren als *System Call* implementiert ist, und somit auf Kernebene ausgeführt wird, kann es nicht passieren, dass mehrere Prozessoren zufällig gleichzeitig einen freien Semaphor verwenden.

6.4.3 MPStuBS-Code-Referenzen

- Die Klasse Spinlock implementiert *Spinlocks*.

6.5 Anpassung des Schedulers

Da alle Prozessoren in MPStuBS gleichberechtigt sein sollen, muss es der Scheduler allen ermöglichen

- Prozesse anzumelden,
- Prozesse zu wechseln,
- abgearbeitete Prozesse zu beenden,
- und nicht benötigte Prozesse zu löschen.

Um das zu realisieren, muss lediglich der Zugriff auf die globale *Readylist* synchronisiert werden. Da aber jeder Scheduleraufruf auf Kernebene ausgeführt wird, und der Kerneintritt über ein *Spinlock* gesichert ist (siehe Abschnitt 6.4), werden keine weiteren Synchronisationsmechanismen benötigt.

6.5.1 *Idle-Thread*

Sollte ein Prozessor, nachdem er den aktuell laufenden Prozess abgearbeitet hat, keinen weiteren laufbereiten Prozess in der *Readylist* finden, muss er in einem *Idle*-Zustand warten, bis neue Prozesse angemeldet oder blockierte Prozesse reaktiviert wurden. Dieser *Idle*-Zustand ist eine Schleife, in der sich der Prozessor schlafen legt, bis er von einem Interrupt aufgeweckt wird. Dann überprüft er, ob sich wieder ein Prozess in der *Readylist* befindet, und führt diesen gegebenenfalls aus. Sollte kein Prozess vorhanden sein, beginnt der Schleifendurchlauf erneut, und der Prozessor legt sich wieder schlafen.

In einem Mono-Prozessor-System wird diese Schleife im Kontext des zuletzt bearbeiteten Prozesses ausgeführt. In einem Multi-Prozessor-System besteht nun die Gefahr, dass der Prozess, in dessen Kontext ein Prozessor den *Idle-Loop* durchführt, wieder in die Liste eingehängt wird, und dann aber von einem anderen Prozessor ausgeführt wird. Daraufhin würden zwei Prozessoren im Kontext eines Prozesses arbeiten, was nicht sein darf.

Deshalb wurden in MPStuBS eigene, von den Benutzerprozessen unabhängige *Idle-Threads* erzeugt. In einer *Idlelist* befindet sich für jeden vorhandenen Prozessor ein solcher *Thread*. Sollte also kein weiterer laufbereiter Prozess zur Verfügung stehen, holt sich der Prozessor einen *Idle-Thread* aus der *Idlelist* und führt diesen aus. Beim Verlassen des *Idle-Threads* muss darauf geachtet werden, dass der *Idle-Thread* wieder in die *Idlelist* eingereiht wird. Im Folgenden wird genauer auf die *Idle-Loop* eingegangen.

Idle-Loop

Bevor die Schleife betreten werden kann, sollten alle anstehenden Interrupts abgearbeitet sein, und dann Interrupts gesperrt werden, um zu verhindern, dass sich der Prozessor schlafen legt, obwohl ein Interrupt einen neuen Prozess anmelden will. Wurden alle Interrupts bearbeitet, und es liegt kein weiterer Prozess vor, betritt der Prozessor die Schleife und legt sich schlafen. Direkt davor lässt er Interrupts wieder zu. Wird der schlafende Prozessor von einem Interrupt aufgeweckt, sperrt er Interrupts wieder, um dann ungestört zu überprüfen, ob ein neuer Prozess angemeldet wurde. Ist dies der Fall, wird zu dem neuen Prozess gewechselt, und der *Idle-Thread* in die *Idlelist* eingehängt, ansonsten beginnt der Schleifendurchlauf erneut.

Dieses Prinzip funktioniert auf einem Mono-Prozessor-System einwandfrei. In einem Multi-Prozessor-System müssen zusätzlich alle Zugriffe auf die *Readylist* über ein *Spinlock* synchronisiert werden, da parallel ein anderer Prozessor auf diese Liste zugreifen könnte. Da es nicht möglich ist, Interrupts global zu sperren, kann es passieren, das sich ein Prozessor schlafen legt, obwohl ein anderer soeben einen neuen Prozess angemeldet hat. Dann erkennt der Prozessor erst bei seinem nächsten Interrupt, dass ein laufbereiter Prozess zur Verfügung steht. Da aber alle Prozessoren regelmäßig von *Timer*-Interrupts unterbrochen werden, sind die Leistungsverluste gering und werden hier akzeptiert.

6.5.2 MPStuBS-Code-Referenzen

- Die Klasse Scheduler wurde erweitert.
- Die Klasse Idle implementiert *Idle-Threads*.

6.6 Zusammenfassung

Der BSP liest die *Floating Pointer Structure* und die *MP Configuration Table* aus, um die vohandenen Hardware-Komponenten und deren Konfiguration zu erkennen. Danach schickt er Inter-Prozessor-Interrupts, indem er das ICR des lokalen APICs beschreibt, um zusätzliche Prozessoren gegebenenfalls zu wecken.

Lokale und I/O-APICs werden über ihren Registersatz programmiert. Bei lokalen APICs ist dieser in den Speicher gemappt, und einzelne Register können direkt über Speicheradressen beschrieben werden. Bei I/O-APICs sind nur zwei Register in den Speicher gemappt, und alle anderen können über diese indirekt beschrieben werden. Um Interrupts statisch oder dynamisch auf alle Prozessoren zu verteilen, müssen die *I/O Redirection Table*-Register des I/O-APICs beschrieben werden.

Die zur Synchronisation der Prozessoren verwendeten *Spinlocks* werden über atomare Operationen implementiert.

Der Scheduler stellt für jeden Prozessor einen *Idle-Thread* zur Verfügung, um zu gewährleisten, dass alle Prozessoren in einem eigenen Thread warten, falls keine Arbeit ansteht.

Kapitel 7

Zusammenfassung und Ausblick

In diesem Kapitel wird nochmal zusammengefasst, was in dieser Arbeit erreicht wurde (Abschnitt 7.1), und an welchen Stellen man noch tiefer ins Detail gehen könnte (Abschnitt 7.2).

7.1 Zusammenfassung der Ergebnisse

In der vorliegenden Studienarbeit wurde die multiprozessorfähige Variante MPStuBS des Lehrbetriebssystems OOStuBS entwickelt. MPStuBS basiert auf einer *Multiple Instruction Multiple Data*, *Shared Memory*, *Tightly Coupled*, *Symmetric* Multi-Prozessor-Architektur. Über die verteilte APIC-Architektur, bestehend aus einem lokalen APIC pro Prozessor, und mindestens einem I/O-APIC ist es möglich, Interrupts auf die verschiedenen Prozessoren statisch oder dynamisch zu verteilen.

Nach dem Systemstart führt der *Bootstrap*-Prozessor das Betriebssystem aus. Aus vom BIOS zur Verfügung gestellten Datenstrukturen liest er nötige Informationen über die vorhandene Hardware und deren Konfiguration. Sollten mehrere Prozessoren vorhanden sein, weckt er diese, indem er Inter-Prozessor-Interrupts verschickt. Danach werden die APICs, die noch in einem abwärtskompatiblen Modus arbeiten, der alle Interrupts an den BSP schickt, so programmiert, dass alle Interrupts auf die vorhandenen Prozessoren verteilt werden.

Die parallel arbeitenden Prozessoren werden auf Benutzerebene über Semaphoren, und auf Kernebene über Spinlocks synchronisiert.

Zeitscheiben-Scheduling garantiert die gleichberechtigte Behandlung der abzuarbeitenden Prozesse. Sollte keine Arbeit anstehen, befinden sich alle Prozessoren in *Idle-Threads*.

7.2 Ausblick

Das entwickelte Betriebssystem MPStuBS soll die abzuarbeitende Last auf alle Prozessoren gleichmäßig verteilen, d.h.die anstehenden Interrupts sollen gleichmäßig auf alle vorhandenen Prozessoren verteilt werden. Interrupts können allerdings nur explizit an einen bestimmten Prozessor, oder an den Prozessor, der mit der geringsten Priorität arbeitet, verschickt werden. Da in MPStuBS, wie auch schon in OOStuBS, kein Konzept implementiert ist, das Prozessen Prioritäten zuordnet, arbeiten alle Prozessoren mit der gleichen, nie veränderten Priorität. Deshalb werden die Interrupts nach einem *Round Robin*-Verfahren auf alle Prozessoren verteilt werden [4]. Dieses Verfahren führt in MPStuBS allerdings nicht zu der gewünschten, gerechten Verteilung. Es wäre also interessant, das Prozesskonzept von MPStuBS noch um Prioritäten zu erweitern.

In MPStuBS darf sich, wie in Abschnitt 5.2 definiert, nur ein Prozessor im Kern befinden. Sollte viel Kernaktivität anstehen, führt das dazu, dass sich mit Ausnahme des Prozessors, der im Kern ist, alle Prozessoren in einem *Spinlock* aufhalten und darauf warten, dass der Kern wieder frei wird. Es wird also wenig Arbeit parallel ausgeführt, und die Gesamtleistung nimmt trotz mehrerer Prozessoren nur geringfügig zu. Indem man den Kern in einzelne, unabhängige Module aufteilt, und es erlaubt, dass sich in jedem Modul ein Prozessor aufhält, könnte die Gesamtleistung deutlich verbessert werden. Die zum Test von MPStuBS verwendete Anwendung „Pacman“ bringt allerdings so wenig Last mit sich, dass sich die Prozessoren hauptsächlich in *Idle-Threads* befinden. Um also die Leistungssteigerung von MPStuBS gegenüber OOStuBS zu prüfen, wäre eine lastintensivere Anwendung nützlich.

Um das Betriebssystem zu testen, wurden mehrere Hardware-Plattformen und der multiprozessorfähige Emulator Bochs (Version 2.3.5) verwendet. Leider sind in diesem Emulator weder die *Floating Pointer Structure*, noch die *MP Configuration Table* vollständig implementiert. Auch das Verteilen der Interrupts im *Lowest Priority Delivery Mode* wird nicht unterstützt. Es bietet sich deshalb an, gegebenenfalls andere multiprozessorfähige Emulatoren zu verwenden.

Anhang A

Quellcode

Alle für MPStuBS neu erstellten und aus OOStuBS veränderten Dateien werden im Folgenden aufgelistet:

smp_detect.h:

```
/*****************************************************************************/
/* Betriebssysteme                                                           */
/*---------------------------------------------------------------------------*/
/*                                                                           */
/*                          S M P _ D E T E C T                              */
/*                                                                           */
/*---------------------------------------------------------------------------*/
/* In smp_detect.h werden Strukture definiert, die in smp_detect.cc          */
/* verwendet werden                                                          */
/*****************************************************************************/

#ifndef __smp_detect_include__
#define __smp_detect_include__

#define  COMPATIBILITY_PIC      1 /* = 1 -> IMCR vorhanden (PIC comp. mode)   */
#define  COMPATIBILITY_VIRTWIRE 0 /* = 0 -> IMCR nicht vorhanden (Virtual Wire comp. mode) */

/* Definition der MP_Floating_Pointer_Structure: */
struct mp_floating_pointer_structure {
  char signature[4];        /* "_MP_"  (ID Signatur)                          */
  unsigned long physptr;    /* Adresse der MP_Configuration_Table             */
  unsigned char length;     /* Länge der Floating_Pointer_Structure (16-Byte units)    */
  unsigned char spec_rev;   /* MP Version (0x01 = 1.1; 0x04 = 1.4)            */
  unsigned char checksum;   /* Checksumme (Gesamtsumme muss 0 sein)           */
  unsigned char feature1;   /* 0 = MP conf tbl; (!=0) = default config        */
  unsigned char feature2;   /* Bit7 set = IMCR|PIC else virt. wire            */
  unsigned char feature3;   /* reserviert (0)                                 */
  unsigned char feature4;   /* reserviert (0)                                 */
  unsigned char feature5;   /* reserviert (0)                                 */
};
#define MPFPS_ID_SIGNATURE (('_'<<24)|('P'<<16)|('M'<<8)|'_')
#define MPFPS_SIZE sizeof(struct mp_floating_pointer_structure)

/* Definition des MP Configuartion Table Headers */
struct mp_config_table_header {
  char signature[4];         /* "PCMP" (Signatur)                             */
  unsigned short length;     /* Länge der Tabelle (in bytes)                  */
  char spec_rev;             /* Version 1=v1.1, 4=v1.4                        */
  char checksum;             /* Gesamtsumme (incl.checksum) muss 0 sein       */
  char oemid[8];             /* id des sys manufacturer                       */
  char productid[12];        /* Produktfamilie                                */
  unsigned long oemptr;      /* ptr zu einer OEM config tbl (0 falls nicht vorhanden)  */
```

```
  unsigned short oemsize;    /* Größe der OEM cfg tbl (in bytes)                    */
  unsigned short count;      /* Anzahl der Einträge in der Basis Tabelle            */
  unsigned long lapic;       /* Adresse des lokalen APICs                           */
  unsigned short exttbllen;  /* Größe der extended entries (in bytes)               */
  char exttblchksum;         /* Checksumme für die extended table                   */
  char reserved;             /* reserviert (0)                                      */
};
#define MPCT_SIGNATURE "PCMP"

/* mögliche Einträge in der MPCT-Tabelle */
#define MPT_PROCESSOR   0
#define MPT_BUS         1
#define MPT_IOAPIC      2
#define MPT_IOINT       3
#define MPT_LINT        4

/* Prozessor Einträge in der MP Config Table */
struct mpt_config_processor {
  unsigned char type;           /* Eintrag Typ (=0)                                */
  unsigned char lapicid;        /* ID des lokalen APIC                             */
  unsigned char lapicver;       /* APIC Versionen                                  */
  unsigned char cpuflag;        /* Bit 0(EN):en-/disable; Bit 1(BP):BSP/AP         */
  unsigned long cpusignature;   /* Prozessortyp (e.g. Pentium)                     */
  unsigned long featureflags;   /* CPUID feature value                             */
  unsigned long reserved[2];
};
#define CPU_ENABLED         1   /* Prozessor ist verfügabar                        */
#define CPU_BOOTPROCESSOR   2   /* Prozessor ist der BSP                           */

#define CPU_MODEL_MASK    0xF0
#define CPU_FAMILY_MASK   0xF00

/* Bus Einträge in der MP Config Table */
struct mpt_config_bus {
  unsigned char type;       /* Eintrag Typ (=1)                                    */
  unsigned char busid;      /* vom BIOS zugewiesen                                 */
  unsigned char bustype[6]; /* ID String                                           */
};

/* IO APIC Einträge in der MP Config Table */
struct mpt_config_ioapic {
  unsigned char type;       /* Entrag Typ (=2)                                     */
  unsigned char apicid;     /* ID dieses I/O APIC                                  */
  unsigned char apicver;    /* Version dieses I/O APIC                             */
  unsigned char flags;      /* BIT 0(EN):en-/disable                               */
  unsigned long apicaddr;   /* Basis Adresse dieses I/O APIC                       */
};
#define IOAPIC_ENABLED    0x01

/* Interrupt Einträge in der MP Config Table */
struct mpt_config_int {
  unsigned char type;       /* Eintrag Typ (=3(I/O) or =4(local))                   */
  unsigned char irqtype;
  unsigned short irqflag;   /* Bit 0&1(PO): Polarity
                               Bit 2&3(EL): Trigger mode                            */
  unsigned char srcbus;     /* source bus ID */
  unsigned char srcbusirq;  /* source bus IRQ signal number                         */
  unsigned char dstapic;    /* ID of connected I/O APIC (0xff=all)                  */
  unsigned char dstirq;     /* INTINn pin to which the signal is connected          */
};

/* stellt fest ob es sich um ein Multi Prozessor System handelt */
int smp_detect(void); /* gibt 1 zurück falls es sich um ein MP System handelt, ansonsten 0 */

#endif
```

smp_detect.cc:

```
/*****************************************************************************/
/* Betriebssysteme                                                           */
/*---------------------------------------------------------------------------*/
/*                                                                           */
/*                          S M P _ D E T E C T                              */
/*                                                                           */
/*---------------------------------------------------------------------------*/
/* In smp_detect.cc werden die Funktionen definiert, die aufgerufen werden   */
/* um zu prüfen ob es sich um ein Multiprozessorsystem handelt.              */
/*****************************************************************************/

/* INCLUDES */
#include "smp/smp_detect.h"
#include "device/cgastr.h"
#include "smp/smp.h"
#include "smp/lapic.h"

/* GLOBALE VARIABLEN */
extern CGA_Stream kout;
extern LAPIC lapic;

extern void smp_delay(int time);

extern unsigned int num_processors_online; /* Anzahl der laufenden prozessoren */

/* Prozessoren */
unsigned int num_processors_present = 0; /* Anzahl von Prozessoren */
unsigned char BSP_id = 0; /* ID des BootStrapProcessor */
unsigned char present_cpu_ID[MAX_NR_CPUS]; /* the CPUs local APIC ID of all CPUs */
int lapic_version[MAX_NR_CPUS];   // Version des lokalen APIC
int compatibility_mode; /* siehe defines COMPATIBILITY_..(in smp_detect.h) */

/* METHODEN */

/* berechnet die Checksumme eines MP Blocks,
 * *mp = Startadresse des Blocks
 * len = länge des Blocks
 * Rückgabewert = 0 falls OK, ansonsten != 0 */
int mpb_checksum(unsigned char *mp, int len)
{
  int sum = 0;
  while (len--)
    sum += *mp++;
  return sum & 0xFF;
}

/* gibt den Namen eines Prozessors zurück */
static char *mpc_family(int family, int model)
{
  static char *model_defs[] = {
    "80486DX", "80486DX",
    "80486SX", "80486DX/2 or 80487",
    "80486SL", "Intel5X2(tm)",
    "Unknown", "Unknown",
    "80486DX/4"
  };
  if (family == 0x04 && model < 9) /*100 */
    return model_defs[model];
  else if (family == 0x5) { /*101 */
    if (model == 4)
      return ("Pentium with MMX technology");
    else
      return ("Pentium");
  }
  else if (family == 0x6) { /*110 */
    if (model == 1)
      return ("Pentium Pro");
    else if (model == 3)
```

```
      return ("Pentium II (model 3)");
    else if (model == 5)
      return ("Pentium II (model 5) or Celeron");
    else if (model == 6)
      return ("Celeron");
    else if (model == 7)
      return ("Pentium III (model 7)");
    else if (model == 8)
      return ("Pentium III (model 8) or Celeron");
    else if (model == 10)
      return ("Pentium III Xeon (model A)");
    else
      return ("P6 family");
  }
  else if (family == 0x0F && model == 0x0F) /*111 */
    return ("Special controller");
  else
    return ("Unknown CPU");
}

/* liest die MP Configuration Table
 * mpcth = pointer auf den Header der Tabelle
 * Rückgabewert = Anzahl Prozessoren */
static int smp_read_mpct(struct mp_config_table_header *mpcth)
{
  int count = sizeof(*mpcth);
  unsigned char *mpt = ((unsigned char *) mpcth) + count; /* pointer auf die Tabelle */

  // Überprüfen der Signatur
  int i;
  for (i = 0; i < 4; i++)
    if ((mpcth->signature)[i] != MPCT_SIGNATURE[i]) {
      kout << "PANIC: SMP mptable: bad signature [" << mpcth->signature[0] << mpcth->signature[1] <<
      mpcth->signature[2] << mpcth->signature[3] << "]" << endl;
      return 1;
    }

    // Überprüfen der Checksumme
    if (mpb_checksum((unsigned char *) mpcth, mpcth->length)) {
      kout << "PANIC: SMP mptable: checksum error!" << endl;
      return 1;
    }
    else {
      kout << "MPCT Checksum OK!" << endl;
    }

    // Auslesen des restlichen MPCT Headers
    kout << "# of table Entries: " << mpcth->count << endl;

    /* jetzt werden die configuration blocks bearbeitet */
    // solange weniger Bytes (count) gelesen wurden als die Groesse der MPCT (mpct->length) bleiben
// wir in der while-schleife
    while (count < mpcth->length) {
      switch (*mpt) {
        case MPT_PROCESSOR:
          {
            struct mpt_config_processor *m = (struct mpt_config_processor *) mpt;
            if (m->cpuflag & CPU_ENABLED) {
              num_processors_present++;
              kout << "Processor #" << num_processors_present << "(ID:"
                   << (unsigned short)m->lapicid << ") "
                   << (mpc_family((m->cpusignature & CPU_FAMILY_MASK) >> 8,
                      (m->cpusignature & CPU_MODEL_MASK) >> 4))
                   << " APIC version "
                   << (unsigned short)m->lapicver << endl;
              /* Auslesen der Struktur */
              if (m->featureflags & (1 << 9))
                kout << "    Internal APIC present" << endl;
              else
                kout << "    no Internal APIC present" << endl;
                if (m->cpuflag & CPU_BOOTPROCESSOR) {
```

```
                kout << "    Bootup CPU" << endl;
                BSP_id = m->lapicid;
              }

            if (num_processors_present > MAX_NR_CPUS)
              kout << "Processor ID: " << m->lapicid << " unused. (Max " << MAX_NR_CPUS
                   << " processors)." << endl;
            else {
              present_cpu_ID[num_processors_present - 1] = m->lapicid;
              lapic_version[m->lapicid] = m->lapicver;
            }
          }
          mpt += sizeof(*m);      // mpt auf die Adresse nach der MPT_PROCESSOR Struktur setzen
          count += sizeof(*m); // count um die Anzahl gelesener Bytes erhöhen
          break;
        }
      /* Die Informationen in den anderen Strukturen sind nicht so wichtig, wir lesen sie also
         nicht aus */
      case MPT_BUS:
        {
          struct mpt_config_bus *m = (struct mpt_config_bus *) mpt;
          mpt += sizeof(*m);
          count += sizeof(*m);
          break;
        }
      case MPT_IOAPIC:
        {
          struct mpt_config_ioapic *m = (struct mpt_config_ioapic *) mpt;
          mpt += sizeof(*m);
          count += sizeof(*m);
          break;
        }
      case MPT_IOINT:
        {
          struct mpt_config_int *m = (struct mpt_config_int *) mpt;
          mpt += sizeof(*m);
          count += sizeof(*m);
          break;
        }
      case MPT_LINT:
        {
          struct mpt_config_int *m = (struct mpt_config_int *) mpt;
          mpt += sizeof(*m);
          count += sizeof(*m);
          break;
        }
    }
  }
  return num_processors_present;
}

/* sucht im übergebenen Speicherbereich nach der MP_Floating_Pointer_Structure
  (die vorhanden sein muss falls es sich um ein smp-System handelt: */
static int smp_search_config(unsigned long base, unsigned long length) {
  unsigned long* bp = (unsigned long*) base;
  struct mp_floating_pointer_structure* mpfps;

  kout << "Scanning for MP Floating Pointer Structure @ " << bp << " for "
       << length << " Bytes." << endl;

  while (length > 0) {
    if (*bp == MPFPS_ID_SIGNATURE) {
      kout << "_MP_ found @ " << bp << endl;
      mpfps = (struct mp_floating_pointer_structure *) bp;
      if (mpfps->length == (MPFPS_SIZE / 16)                  // prüft ob die Länge der Struktur passt
          && !mpb_checksum((unsigned char *) bp, MPFPS_SIZE)  // prüft die Checksumme
          && (mpfps->spec_rev == 1 || mpfps->spec_rev == 4))  // prüft ob es sich um einen gültige
      {                                                       // Version der MP Spec handelt
        kout << "FPS Checksum OK!" << endl;
        kout << "Intel MultiProcessor Specification v1." << (unsigned short)mpfps->spec_rev << endl;
```

```
      // Auslesen der Struktur (wichtig ist für uns nur der Compatibility Mode)
        compatibility_mode = mpfps->feature2 & (1 << 7);
if (compatibility_mode == COMPATIBILITY_VIRTWIRE)
  kout << "    Virtual Wire compatibility mode" << endl;
        else
  kout << "    PIC compatibility mode (IMCR present)" << endl;

      /* Prüfen ob es sich um eine Standart Konifiguration handelt (ist bei uns nicht der Fall) */
        if (mpfps->feature1 != 0) {  // wenn feature1 gleich 0 ist handelt es sich um eine default
                                     // konfiguration
          kout << "    System has a default configuration" << endl;
        }
        else {
          if (mpfps->physptr != 0x0) { // wenn physptr ungleich 0 ist liegt eine MPCT vor
            kout << "    MP Configuration Table exists!" << endl;
            smp_read_mpct((struct mp_config_table_header *)mpfps->physptr);
          }
        }
        /* Die erste gefundene Konfiguration wird verwendet. */
        return 1;
      }
    }
    bp += 4;       // die Struktur ist 16 Byte groß, die nächste zu prüfende Adresse ist also vier
                   // longs größer als die letzte
    length -= 16;  // die ersten 16 Byte wurden geprüft, length kann also um 16 reduziert werden
  }
  return 0;        // falls die Signatur nicht gefunden wurde
}
/* stellt fest ob es sich um ein Multi Prozessor System handelt */
int smp_detect(void) {
  unsigned short memsize = 0;
  /*
  * Die Struktur kann an drei verschiedenen Stellen im Speicher liegen:
  * 1) im obersten 1KB von EBDA
  * 2) im obersten 1KB vom Basis RAM (639K-640K or 511K-512K)
  * 3) im BIOS ROM (0x0f0000-0x0fffff)
  */
  memsize = (*(unsigned short *) (0x413));
  kout << "System base Memory: " << memsize << endl;

  if (smp_search_config((*(unsigned short *) (0x40E)), 1024) ||
      smp_search_config(memsize * 1024, 1024) ||
      smp_search_config(0x0F0000, 64 * 1024)) {
    return 1;
  }
  else {
    kout << "MP Floating Pointer Structure not found!-->This is a single
         processor machine!" << endl;
    num_processors_online = 1;
    return 0;
  }
}
```

smp_activate.h:

```
/*****************************************************************************/
/* Betriebssysteme                                                           */
/*---------------------------------------------------------------------------*/
/*                                                                           */
/*                       S M P _ A C T I V A T E                             */
/*                                                                           */
/*---------------------------------------------------------------------------*/
/* In smp_activate.h werden werden alle Methoden deklariert die nötig sind   */
/* um Application-Prozessoren zu starten                                     */
/*****************************************************************************/

#ifndef __smp_activate_include__
#define __smp_activate_include__
```

```
/* startet alle Application-Prozessoren, und konfiguriert den IO-APIC */
void smp_init(void);

/* meldet dem BSP dass man aufgewacht ist */
void smp_callin(void);

/* verzögert um time * 10 msecs */
void smp_delay(int time);

#endif
```

smp_activate.cc:

```
/******************************************************************************/
/* Betriebssysteme                                                            */
/*----------------------------------------------------------------------------*/
/*                                                                            */
/*                       S M P _ A C T I V A T E                              */
/*                                                                            */
/*----------------------------------------------------------------------------*/
/* In smp_activate.cc werden werden alle Methoden definiert die nötig sind   */
/* um Application-Prozessoren zu starten                                      */
/******************************************************************************/

/* INCLUDES */
#include "device/cgastr.h"
#include "smp/smp.h"
#include "machine/io_port.h"
#include "smp/timer8254.h"
#include "smp/smp_activate.h"
#include "smp/io_apic.h"
#include "smp/lapic.h"
#include "machine/spinlock.h"

/* GLOBALE VARIABLEN */
extern CGA_Stream kout;
extern IO_APIC io_apic;
extern LAPIC lapic;
extern Spinlock kout_lock;

extern unsigned int num_processors_present;
extern unsigned char BSP_id;
extern unsigned char present_cpu_ID[MAX_NR_CPUS];
extern int lapic_version[MAX_NR_CPUS];

/* VARIABLEN  */
unsigned int num_processors_online = 0; /* Anzahl der laufenden prozessoren */

static volatile unsigned long cpu_callout[MAX_NR_CPUS]; /* wird vom BSP gesetzt um einen AP zu
                                                           aktivieren */
static volatile unsigned long cpu_callin[MAX_NR_CPUS]; /* wird von einem aktiven AP gesetzt um
                                                          dem BSP zu sagen das man läuft */

/* verzögert um time * 10 msecs */
void smp_delay(int time)
{
  int i;
  for (i = 0; i < time; i++)
    wait_8254_wraparound();
  return;
}

/* aktiviert die CPU mit der ID id, gibt bei erfolg 1, ansonsten 0 zurück */
int do_boot_cpu(int id)
{
  int result = 0;
  unsigned long send_status;
  int timeout, j; /* wird in STARTUP loop verwendet */
  unsigned long start_eip = (unsigned long)0x1000;
```

```
  kout << "Booting processor #" << id << " at EIP " << (void *)start_eip << endl;

  /* Start des Aufweck-Prozesses des über id adressierten Prozessors */
  kout << "Setting warm reset code and vector" << endl;

  /* setzen des BIOS shutdown code auf warm start (*CMOS0xf=0xa) */
  outb(0xf, 0x70); /* the CMOS is controlled by the RTC (port 70) */
  outb(0xa, 0x71);

  /* setzen des reset vector */
  *((volatile unsigned short *) 0x469) = start_eip >> 4;
  *((volatile unsigned short *) 0x467) = start_eip & 0xf;

/*---------- los gehts -------   */
  /* Status ist jetzt ok */
  send_status = 0;
  cpu_callout[id] = 0;

  /* Starten der IPI sequenz... */
  kout << "Asserting INIT" << endl;

  /* anlegen von INIT */
  lapic.setIRQdest(id);
  lapic.set_APIC_ICR(APIC_DEST_LEVELTRIG | APIC_DEST_ASSERT | APIC_DEST_DM_INIT);

  smp_delay(2);
  kout << "Deasserting INIT" << endl;

  /* zurücknehmen von INIT */
  lapic.setIRQdest(id);
  lapic.set_APIC_ICR(APIC_DEST_LEVELTRIG | APIC_DEST_DM_INIT);

  /*
  Sollen STARTUP IPIs geschickt werden??
  falls wir einen 82489DX haben, brauchen wir keine STARTUP IPIs
  */
  if ((lapic_version[id] & 0xF0) != APIC_VER_82489DX) {
    /* Beginnen der STARTUP IPI schleife */
    for (j = 0; (j < 2); j++) {
      kout << "Sending STARTUP #" << (j + 1) << endl;

      /* schicken des STARTUP IPI */
      lapic.setIRQdest(id);
      lapic.set_APIC_ICR(APIC_DEST_DM_STARTUP | (start_eip >> 12));

      timeout = 0;
      kout << "Waiting for send to finish...";
      /* warten ob STARTUP IPI ok? */
      do {
        kout << "+";
        smp_delay(1);
        timeout++;
        send_status = lapic.read_icr() & 0x1000;
      } while (send_status != 0 && (timeout < 10));

      if (timeout == 10)
      kout << "timed out" << endl;
      else
      kout << "send OK" << endl;

      if (send_status == 0)
      break; /* alles OK */
    } /* for */

    kout << "After Startup" << endl;
  }
  else
    kout << "No STARTUP IPI required." << endl;

  if (send_status == 0) {
```

```
    kout << "STARTUP IPI OK with CPU #" << id << endl;
    /* den APs erlauben die initialisierung zu starten */
    kout << "Before Callout CPU #" << id << endl;
    cpu_callin[id] = 0;
    cpu_callout[id] = 1;

    /* Ein anderer Prozessor ist jetzt wach und könnte ebenfalls etwas auf dem Bildschirm ausgeben,
       deshalb werden Bildschirmausgaben hier noch mit einem Spinlock gesichert */
    kout_lock.lock();
    kout << "After Callout CPU #" << id << endl;
    kout_lock.unlock();

    for (timeout = 0; timeout < 500; timeout++) {
      if (cpu_callin[id])
        break; /* cpu hat gebooted  */
      smp_delay(1);
    }
    if (cpu_callin[id]) {
      kout_lock.lock();
      kout << "OK" << endl;
      kout_lock.unlock();
      result = 1;
      kout_lock.lock();
      kout << "CPU #" << id << " has booted" << endl;
      kout_lock.unlock();
    }
    else /* prozessor gibt keine antwort */
      kout << "CPU #" << id << " Not responding" << endl;
  }
  else /* STARTUP IPI fehlgeschlagen */
    kout << "STARTUP IPI failed with CPU #" << id << endl;
  return result;
}

/* schickt allen APs IPIs um sie zu booten */
void smp_boot_cpus(void)
{
  unsigned int i;

  /* alle vorhandenen CPUs werden gestartet */
  for (i = 0; i < num_processors_present; i++) {
  /* Don't start the boot CPU! */
    if (present_cpu_ID[i] == BSP_id || /* BSP */
      do_boot_cpu(present_cpu_ID[i])) { /* or AP is online */
      num_processors_online++;
      kout_lock.lock();
      kout << num_processors_online << " processor/s is/are online!" << endl;
      kout_lock.unlock();
    }
else
      kout << "CPU #" << (unsigned short)present_cpu_ID[i] << " is not responding." << endl;
  }

}

/* Wird vom Bootstrap Prozessor aufgerufen und aktiviert die Application-Prozessoren */
void smp_init(void)
{
  smp_boot_cpus();  // weckt alle vorhandenen Prozessoren auf
  lapic.set_logical_destination_address(0x01);  // setzt die logische Adresse des BSP
  io_apic.setup_IO_APIC();  // konfiguriert den IO-APIC so, dass er Interrupts an die lokalen APICs
                            // verteilt
}

/***************** Funktionen für die APs ***************************/

/* die erste Methode, die ein AP ausführ, nachdem er aufgeweckt wurde,
   meldet dem BSP dass man aufgewacht ist */
void smp_callin(void)
{
  int cpuid;
```

```
  unsigned long timeout = 0;

  /* wer bin ich? */
  cpuid = lapic.get_processor_id();

  kout_lock.lock();
  kout << "CPU #" << cpuid << " is waiting for CALLOUT" << endl;
  kout_lock.unlock();

  while (timeout++ < 10000000) {
    /* ist die boot CPU fertig mit der STARTUP sequenz?
    wurden wir vom BSP gerufen? */
    if (cpu_callout[cpuid])
      break;
    smp_delay(1);
  }

  if (timeout++ > 10000000) {
    kout << "BUG: CPU #" << cpuid << " started up but did not get a callout!" << endl;
    while(true){}
  }

  kout_lock.lock();
  kout << "CPU #" << cpuid << " called out!" << endl;
  kout_lock.unlock();

  /*
   * die boot CPU ist mit init fertig und wartet in cpu_callin
   * bis wir fertig sind. wir können jetzt die cpu einrichten,
   * wir fangen mit dem APIC an (das ist meistens unötig)
   */

  lapic.enable_local_APIC(); // aktiviert und konfiguriert den lokalen APIC des APs
  lapic.set_logical_destination_address(0x02); // setzt die logischen Adresse des APs

  /* erlaubt der boot CPU weiter zu machen */
  kout_lock.lock();
  kout << "CPU #" << cpuid << " sets callin bit" << endl;
  kout_lock.unlock();
  cpu_callin[cpuid] = 1;
}
```

tim8254.h:

```
/*****************************************************************************/
/* Betriebssysteme                                                           */
/*---------------------------------------------------------------------------*/
/*                                                                           */
/*                          T I M E R   8 2 5 4                              */
/*                                                                           */
/*---------------------------------------------------------------------------*/
/* In timer8254.h werden die Methoden deklariert die benötigt werden um einen*/
/* Prozessor warten zu lassen                                                */
/*****************************************************************************/

#ifndef __timer8254_include__
#define __timer8254_include__

void wait_8254_wraparound(void);
unsigned int get_8254_timer_count(void);

#endif
```

timer8254.cc:

```
/*****************************************************************************/
/* Betriebssysteme                                                           */
/*---------------------------------------------------------------------------*/
/*                                                                           */
/*                          T I M E R   8 2 5 4                              */
/*                                                                           */
/*---------------------------------------------------------------------------*/
/* In timer8254.h werden die Methoden definiert die benötigt werden um einen */
/* Prozessor warten zu lassen                                                */
/*****************************************************************************/

#include "smp/timer8254.h"
#include "machine/io_port.h"


unsigned int get_8254_timer_count(void) {
  unsigned int count;

  outb(0x00, 0x43);
  count = inb(0x40);
  count |= inb(0x40) << 8;

  return count;
}

void wait_8254_wraparound(void) {
  unsigned int curr_count, prev_count = ~0;
  int delta;

  curr_count = get_8254_timer_count();

  do {
    prev_count = curr_count;
    curr_count = get_8254_timer_count();
    delta = curr_count - prev_count;

  } while (delta < 300);
}
```

setup2nd.asm:

```
;*****************************************************************************
;* Betriebssysteme                                                           *
;*---------------------------------------------------------------------------*
;*                                                                           *
;*                            S E T U P  2nd                                 *
;*                                                                           *
;*---------------------------------------------------------------------------*
;* Der Setup-Code liegt an 0x00001000. Er wird nach einem startup ipi noch im *
;* 'Real-Mode' gestartet, so dass zu Beginn auch noch BIOS-Aufrufe erlaubt    *
;* sind. Dann werden jedoch alle Interrupts verboten, die Adressleitung A20   *
;* aktiviert und die Umschaltung in den 'Protected-Mode' vorgenommen. Alles   *
;* weitere uebernimmt der Startup-Code des Systems.                          *
;*****************************************************************************

[GLOBAL setup2nd]

[SECTION .text]

USE16

;
; Segmentregister initialisieren
;
```

```
setup2nd:
  mov   ax,cs ; Daten-, Code- und Stacksegment sollen
  mov   ds,ax ; hierher zeigen.
  mov   ss,ax ; Alle drei Segment duerfen nicht mehr
;
; So, jetzt werden die Interrupts abgeschaltet
;
  cli     ; Maskierbare Interrupts verbieten
  mov   al,0x80 ; NMI verbieten
  out   0x70,al
;
; IDT und GDT setzen
;
  lidt  [idt_48 - setup2nd]
  lgdt  [gdt_48 - setup2nd]
;
; Umschalten in den Protected Mode
;
  mov   ax,1
  lmsw  ax
  jmp   flush_instr

flush_instr:
  jmp   dword 0x08:0x30000
  hlt

gdt:
  dw    0,0,0,0    ; NULL Deskriptor

  dw    0xFFFF   ; 4Gb - (0x100000*0x1000 = 4Gb)
  dw    0x0000   ; base address=0
  dw    0x9A00   ; code read/exec
  dw    0x00CF   ; granularity=4096, 386 (+5th nibble of limit)

  dw    0xFFFF   ; 4Gb - (0x100000*0x1000 = 4Gb)
  dw    0x0000   ; base address=0
  dw    0x9200   ; data read/write
  dw    0x00CF   ; granularity=4096, 386 (+5th nibble of limit)

idt_48:
  dw    0 ; idt limit=0
  dw    0,0 ; idt base=0L

gdt_48:
  dw    0x18 ; GDT Limit=24, 3 GDT Eintraege
  dd    0x1000 + gdt - setup2nd; Physikalische Adresse der GDT
```

startup2nd.asm

```
;*****************************************************************************
;* Betriebssysteme                                                           *
;*---------------------------------------------------------------------------*
;*                                                                           *
;*                     S T A R T U P 2nd . A S M                             *
;*                                                                           *
;*---------------------------------------------------------------------------*
;* 'startup' ist der Eintrittspunkt des eigentlichen Systems. Die Umschaltung *
;* in den 'Protected Mode' ist bereits erfolgt. Es wird alles vorbereitet,   *
;* damit so schnell wie moeglich die weitere Ausfuehrung durch C-Code erfol- *
;* gen kann.                                                                 *
;*****************************************************************************

;
;   System
;
[GLOBAL startup2nd]

[EXTERN main2nd]
[EXTERN guardian]
```

```
[EXTERN idt_descr]

[SECTION .text2nd]

USE32

startup2nd:

; GCC-kompilierter Code erwartet das so.
  cld

; Globales Datensegment

  mov   ax,0x10
  mov   ds,ax
  mov   es,ax
  mov   fs,ax
  mov   gs,ax

; Stack festlegen

  mov   ss,ax
  mov   esp,init_stack+4096
;
; Unterbrechungsbehandlung sicherstellen
;
  lidt  [idt_descr]

; Aufruf des C-Codes

  call  main2nd ; C/C++ Level System
  hlt


[SECTION .bss]

init_stack:
  resb  4096
```

lapic.h:

```
/******************************************************************************/
/* Betriebssysteme                                                            */
/*----------------------------------------------------------------------------*/
/*                                                                            */
/*                                L A P I C                                   */
/*                                                                            */
/*----------------------------------------------------------------------------*/
/* In lapic.h wird die Klasse LAPIC deklariert                                */
/******************************************************************************/

#ifndef __lapicoo_include__
#define __lapicoo_include__

#include "smp/smp.h"

#define APIC_VER_82489DX     0x00 /* 0Xh = 82489DX */
#define APIC_VER_INTEGRATED  0x10 /* 1Xh = local */

/* für ICR */
#define APIC_DEST_DEST       0x00000 /* Destination Shorthand */
#define APIC_DEST_SELF       0x40000
#define APIC_DEST_ALLINC     0x80000
#define APIC_DEST_ALLBUT     0xC0000
#define APIC_DEST_LEVELTRIG  0x08000 /* Trigger Mode */
#define APIC_DEST_ASSERT     0x04000 /* Level */
#define APIC_DEST_BUSY       0x01000 /* Delivery Status */
#define APIC_DEST_LOGICAL    0x00800 /* Destination Mode */
#define APIC_DEST_DM_FIXED   0x00000 /* Delivery Mode */
```

```
#define APIC_DEST_DM_LOWEST  0x00100
#define APIC_DEST_DM_SMI     0x00200
#define APIC_DEST_DM_REMRD   0x00300
#define APIC_DEST_DM_NMI     0x00400
#define APIC_DEST_DM_INIT    0x00500
#define APIC_DEST_DM_STARTUP 0x00600

class LAPIC
 {
private:
  unsigned long lapic_base;
  enum {
    id_reg    = 0x20,   /* ID Register                              */
    eoi_reg   = 0xB0,   /* End of Interrupt Register                */
    icr_reg   = 0x300,  /* Interrupt Command Register 0-31   (r/w) */
    icr2_reg  = 0x310,  /* Interrupt Command Register 32-63  (r/w) */
    ldr_reg   = 0xD0,   /* Logical Destination Register             */
    dfr_reg   = 0xE0,   /* Destination Format Register              */
    tpr_reg   = 0x80,   /* Tast Priority Register                   */
    spiv_reg  = 0xF0    /* Spurious Interrupt Vector Register       */
  };

  /* beschreibt das APIC-register reg mit den daten v */
  void apic_write(unsigned long reg, unsigned long v);

  /* liest das APIC register reg */
  unsigned long apic_read(unsigned long reg);

public:
  /* Konstruktor */
  LAPIC() {
    lapic_base = 0xFEE00000;
  }

  /* liest die prozessor id */
  int get_processor_id(void);

  /* aktiviert den lokalen APIC */
  void enable_local_APIC(void);

  /* bestätigt das ein Interrupt angekommen ist */
  void ack_APIC_irq(void);

  /* schreibt die Ziel-CPU für einen IPI ins ICR */
  void setIRQdest(int cpu_id);

  /* schreibt die daten für einen IPI ins ICR, und schickt damit den IPI ab */
  void set_APIC_ICR(unsigned long data);

  /* liest das ICR Register */
  unsigned long read_icr();

  /* setzt die logische ID des lokalen APICs */
  void set_logical_destination_address(unsigned char data);

 };

#endif
```

lapic.cc:

```
/*****************************************************************************/
/* Betriebssysteme                                                           */
/*---------------------------------------------------------------------------*/
/*                                                                           */
/*                              L A P I C                                    */
/*                                                                           */
/*---------------------------------------------------------------------------*/
/* In lapic.cc werden die Klassenmethoden der Klasse LAPIC definiert         */
/*****************************************************************************/

#include "smp/lapic.h"
#include "device/cgastr.h"

extern CGA_Stream kout;

/* beschreibt das APIC-register reg mit den daten v */
void LAPIC::apic_write(unsigned long reg, unsigned long v) {
  *((volatile unsigned long *) (lapic_base + reg)) = v;
}

/* liest das APIC register reg */
unsigned long LAPIC::apic_read(unsigned long reg) {
  return *((volatile unsigned long *) (lapic_base + reg));
}

/* liest die prozessor id */
int LAPIC::get_processor_id(void) {
  return  ((apic_read(id_reg)>>24)&0x0F);
}

/* aktiviert den lokalen APIC */
void LAPIC::enable_local_APIC(void)
{
  unsigned long value;

  /* löschen der logical destination ID */
   value = apic_read(ldr_reg);
  value &= 0x00ffffff; /* Dest = 0 */
  apic_write(ldr_reg, value);

  /* Task Priority auf 0 setzen, um alle Interrupts zu akzeptieren */
  value = apic_read(tpr_reg);
  value &= 0xffffff00;
  apic_write(tpr_reg, value);

  /* flat delivery mode einstellen */
  value = apic_read(dfr_reg);
  value |= 0xf0000000; /* bits 28-31 = 1111 -> flat mode */
  apic_write(dfr_reg, value);

  /* aktivieren APIC */
  value = apic_read(spiv_reg);
  value |= (1 << 8); /* Enable APIC (bit==1) */
  apic_write(spiv_reg, value);
}

/* bestätigt das ein Interrupt angekommen ist */
void LAPIC::ack_APIC_irq(void) {
  apic_write(eoi_reg, 0);
}

/* schreibt die Ziel-CPU für einen IPI ins ICR */
void LAPIC::setIRQdest(int cpu_id) {
  unsigned long tmp;
  tmp = apic_read(icr2_reg);
  tmp &= 0x00FFFFFF; /* löschen der bits in tmp die die Ziel-CPU für den IPI festlegen */
  tmp |= (cpu_id << 24); /* eintragen der Ziel-CPU für den IPI in tmp */
  apic_write(icr2_reg, tmp); /* tmp wird in den oberen teil (APIC_ICR2) des ICRs geschrieben */
```

```
}

/* schreibt die daten für einen IPI ins ICR, und schickt damit den IPI ab */
void LAPIC::set_APIC_ICR(unsigned long data) {
  unsigned long tmp;
  tmp = apic_read(icr_reg);
  tmp &= ~0xCDFFF; /* löschen aller nötigen bits in tmp */
  tmp |= data;
  /* das beschreiben des unteren teils (APIC_ICR) des ICRs verschickt den IPI */
  apic_write(icr_reg, tmp); /* verschicken des IPI  */
}

void LAPIC::set_logical_destination_address(unsigned char data) {
  unsigned long help = apic_read(ldr_reg);
  unsigned long data_l = ((unsigned long)data) << 24;
  help |= data_l;
  apic_write(ldr_reg,data_l); // das logical destination register mit einer logical destination
                              // beschreiben
}

/* liest das ICR Register */
unsigned long LAPIC::read_icr() {
  return apic_read(icr_reg);
}
```

io_apic.h

```
/*****************************************************************************/
/* Betriebssysteme                                                           */
/*---------------------------------------------------------------------------*/
/*                                                                           */
/*                            I O _ A P I C                                  */
/*                                                                           */
/*---------------------------------------------------------------------------*/
/* In io_apic.h wird die Klasse IO_APIC deklariert                           */
/*****************************************************************************/

#ifndef __io_apicoo_include__
#define __io_apicoo_include__

#define DELIVERY_MODE_FIXED     0
#define DELIVERY_MODE_LOWESTPRI 1
#define DELIVERY_MODE_SMI       2
#define DELIVERY_MODE_NMI       4
#define DELIVERY_MODE_INIT      5
#define DELIVERY_MODE_EXTINT    7

#define DEST_MODE_PHYSICAL 0
#define DEST_MODE_LOGICAL  1

#define APICPOL_LOW     1
#define APICPOL_HIGH    0

#define APICTRIGGER_EDGE  0
#define APICTRIGGER_LEVEL 1

/* The registers of the IO APIC: */
struct IO_APIC_ID_reg {
  int reserved_2:24, ID:4, reserved_1:4;
} __attribute__ ((packed));

struct IO_APIC_VER_reg {
  int version:8, reserved_2:8, entries:8, reserved_1:8;
} __attribute__ ((packed));

struct IO_APIC_ARB_reg {
  int reserved_2:24, arbitration:4, reserved_1:4;
} __attribute__ ((packed));
```

```
struct IO_APIC_route_entry {
  int vector:8, delivery_mode:3,/* see DELIVERY_MODE... defines */
    dest_mode:1,/* see DEST_MODE_... defines */
    delivery_status:1, polarity:1, irr:1, trigger:1,/* see APICTRIGGER_.. defines */
    mask:1,/* 0: enabled, 1: disabled */
    reserved_2:15;
  union {
    struct {
  long reserved_1:24, physical_dest:4, reserved_2:4;
    } physical;
    struct {
      long reserved_1:24, logical_dest:8;
    } logical;
  } dest;
} __attribute__ ((packed));

class IO_APIC
 {
private:
  unsigned long *io_apic_base;
  unsigned long *io_regsel;
  unsigned long *io_win;

  enum {
    io_apic_id_reg = 0x00,      /* ID Register */
    io_apic_ver_reg = 0x01,     /* Versions Register */
    io_apic_arb_reg = 0x02,     /* Arbitration Register */
    io_apic_rdt_reg = 0x10      /* offset für die adressierung des anfangs der redirection table */
  };

  /* schreibt nach IO_REGSEL das zu beschreibende IO-APIC register reg und nach IO_WIN die daten
     value */
  void io_apic_write(unsigned int reg, unsigned long value);

  /* liest das IO-APIC register reg */
  unsigned long *io_apic_read(unsigned int reg);

  /* schreibt entry in die Interrupt Redirection Table des IO-APICs an den Pin pin */
  void io_apic_write_route_entry(unsigned int pin, struct IO_APIC_route_entry entry);

  /* liest die daten des IO-APICs und schreibt die gewuenschten Interrupts in die IR Table */
  void setup_IO_APIC_irqs(void);

  /* schreibt beginnend mit adresse tov den wert c für len bytes in den speicher */
  void *memset(void *tov, int c, int len);

public:
  /* Konstruktor */
  IO_APIC() {
    io_apic_base = (unsigned long *)0xFEC00000;
    io_regsel = (unsigned long *)0xFEC00000;
    io_win = (unsigned long *)0xFEC00010;
  }

  /* stellt den IO-APIC richtig ein */
  void setup_IO_APIC(void);

 };

#endif
```

io_apic.cc:

```
/*****************************************************************************/
/*                            Betriebssysteme                                */
/*---------------------------------------------------------------------------*/
/*                                                                           */
/*                             I O _ A P I C                                 */
/*                                                                           */
/*---------------------------------------------------------------------------*/
/* In io_apic.cc werden die Methoden der Klasse IO_APIC definiert            */
/*****************************************************************************/

#include "smp/io_apic.h"
#include "device/cgastr.h"

extern CGA_Stream kout;

/* schreibt nach IO_REGSEL das zu beschreibende IO-APIC register reg und nach IO_WIN die daten
   value */
void IO_APIC::io_apic_write(unsigned int reg, unsigned long value) {
  *io_regsel = reg; /* über IOREGSEL wird das gewünschte register selektiert */
  *io_win = value; /* über IOWIN wird das gewünschte register beschrieben */
}

/* liest das IO-APIC register reg */
unsigned long* IO_APIC::io_apic_read(unsigned int reg) {
  *io_regsel = reg; /* über IOREGSEL wird das gewünschte register selektiert */
  return (io_win);  /* über IOWIN wird das gewünschte register gelesen*/
}

/* schreibt entry in die Interrupt Redirection Table des IO-APICs an den Pin pin */
void IO_APIC::io_apic_write_route_entry(unsigned int pin, struct IO_APIC_route_entry entry) {
  io_apic_write(io_apic_rdt_reg + 1 + 2 * pin, *(((int *) &entry) + 1));
  io_apic_write(io_apic_rdt_reg + 0 + 2 * pin, *(((int *) &entry) + 0));
}

/* liest die daten des IO-APICs und schreibt die gewuenschten Interrupts in die IR Table */
void IO_APIC::setup_IO_APIC_irqs(void) {
  struct IO_APIC_route_entry entry;

  kout << "init IO-APIC IRQS" << endl;

  /* alle einträge der IO Redirection Table werden auf einen standard wert gelegt, und
     alle interrupts ausmaskiert */
  for(int i = 0;i < 24; i++) {
    memset(&entry, 0, sizeof(entry));
    entry.delivery_mode = DELIVERY_MODE_LOWESTPRI;
    entry.dest_mode = DEST_MODE_LOGICAL;
    entry.mask = 1; // enable = 0, disable = 1
    entry.dest.logical.logical_dest = 0x03; // zum BSP und zum AP
    entry.trigger = APICTRIGGER_EDGE;
    entry.polarity = APICPOL_HIGH;
    entry.vector = 0x21;

    io_apic_write_route_entry(i,entry);
  }

  /* tastaur */
  /* anlegen eines eintrags für die redirection table */
  memset(&entry, 0, sizeof(entry));
  entry.delivery_mode = DELIVERY_MODE_LOWESTPRI;
  entry.dest_mode = DEST_MODE_LOGICAL;
  entry.mask = 0; // enable = 0, disable = 1
  entry.dest.logical.logical_dest = 0x03; // zum BSP und zum AP
  entry.trigger = APICTRIGGER_EDGE;
  entry.polarity = APICPOL_HIGH;
  entry.vector = 0x21;

  io_apic_write_route_entry(1,entry);
```

```
  /* timer */
  /* anlegen eines eintrags für die redirection table */
  memset(&entry, 0, sizeof(entry));
  entry.delivery_mode = DELIVERY_MODE_LOWESTPRI;
  entry.dest_mode = DEST_MODE_LOGICAL;
  entry.mask = 0; // enable = 0, disable = 1
  entry.dest.logical.logical_dest = 0x03; // zum BSP und zum AP
  entry.trigger = APICTRIGGER_EDGE;
  entry.polarity = APICPOL_HIGH;
  entry.vector = 0x20;

  io_apic_write_route_entry(2,entry);

  /* nur für den emulator, in der hardware liegt der timer interrupt am pin 2 */
  /* anlegen eines eintrags für die redirection table */
  /*
  memset(&entry, 0, sizeof(entry));
  entry.delivery_mode = DELIVERY_MODE_LOWESTPRI;
  entry.dest_mode = DEST_MODE_LOGICAL;
  entry.mask = 0; // enable = 0, disable = 1 disable IRQ by default
  entry.dest.logical.logical_dest = 0x03; // zum BSP und zum AP
  entry.trigger = APICTRIGGER_EDGE;
  entry.polarity = APICPOL_HIGH;
  entry.vector = 0x20;

  io_apic_write_route_entry(0,entry);
  */

}

/* schreibt beginnend mit adresse tov den wert c für len bytes in den speicher */
void* IO_APIC::memset(void *tov, int c, int len) {
  register char *to = (char *)tov;

  while (len-- > 0)
    *to++ = c & 0xff;

  return tov;
}

/* stellt den IO-APIC richtig ein und gibt den inhalt der register am bildschirm aus */
void IO_APIC::setup_IO_APIC(void) {

  kout << "SETTING UP IO_APIC" << endl;

  io_apic_write(io_apic_id_reg, 0x02000000);
  struct IO_APIC_ID_reg *id_reg;
  struct IO_APIC_VER_reg *ver_reg;
  struct IO_APIC_ARB_reg *arb_reg;
  id_reg = (struct IO_APIC_ID_reg *)io_apic_read(io_apic_id_reg);
  kout << "IO_APIC_ID: " << (id_reg->ID) << endl;
  ver_reg = (struct IO_APIC_VER_reg *)io_apic_read(io_apic_ver_reg);
  kout << "IO_APIC_VER: " << ver_reg->version << "  Entries: " << ver_reg->entries << endl;
  arb_reg = (struct IO_APIC_ARB_reg *)io_apic_read(io_apic_arb_reg);
  kout << "IO_APIC_ARB: " << arb_reg->arbitration << endl;

  kout << "ENABLING IO-APIC IRQS" << endl;

  setup_IO_APIC_irqs();
}
```

spinlock.h:

```
/*****************************************************************************/
/* Betriebssysteme                                                           */
/*---------------------------------------------------------------------------*/
/*                                                                           */
/*                        S P I N L O C K                                    */
/*                                                                           */
/*---------------------------------------------------------------------------*/
/* Implementierung einer Klasse für Spinlocks                                */
/*****************************************************************************/

#ifndef __Spinlock_include__
#define __Spinlock_include__

class Spinlock
 {
private:
  volatile unsigned int lock_status;
public:

  Spinlock() {
    lock_status = 0;
  }

  void lock() {
    while(__sync_lock_test_and_set(&lock_status,1) == 1) {
    }
  }

  void unlock() {
    lock_status = 0;
  }

  int get_status() {
    return lock_status;
  }
 };

#endif
```

scheduler.h:

```
/*****************************************************************************/
/* Betriebssysteme                                                           */
/*---------------------------------------------------------------------------*/
/*                                                                           */
/*                          S C H E D U L E R                                */
/*                                                                           */
/*---------------------------------------------------------------------------*/
/* Implementierung des Schedulers.                                           */
/*****************************************************************************/

#ifndef __schedule_include__
#define __schedule_include__

#include "thread/dispatch.h"

#include "thread/entrant.h"
#include "object/queue.h"

class Scheduler : public Dispatcher
 {
public:
  bool do_idle;
  bool do_idle2nd;
  Queue readylist;
```

```
  Queue idlelist;
  Scheduler () : do_idle(false), do_idle2nd(false) {}

  void schedule ();
  void ready (Entrant& that);
  void exit ();
  void kill (Entrant& that);
  void resume ();

private:
  void idle ();
 };

#endif
```

scheduler.cc:

```
/*****************************************************************************/
/* Betriebssysteme                                                           */
/*---------------------------------------------------------------------------*/
/*                                                                           */
/*                          S C H E D U L E R                                */
/*                                                                           */
/*---------------------------------------------------------------------------*/
/* Implementierung des Schedulers.                                           */
/*****************************************************************************/

/* INCLUDES */

#include "thread/scheduler.h"
#include "guard/guard.h"
#include "machine/cpu.h"
#include "machine/spinlock.h"
#include "smp/lapic.h"
#include "device/cgastr.h"


/* GLOBALE VARIABLEN */

extern Guard guard;
extern CPU cpu;
extern LAPIC lapic;
extern CGA_Stream kout;
extern Spinlock readylist_lock;

/* IMPLEMENTIERUNG DER METHODEN */

void Scheduler::schedule () {
  Entrant* first;

  if (!active()) {
    readylist_lock.lock(); // zugriff auf die readylist muss gesichert sein, da ein anderer Prozessor
                           // unerwartet aus dem idle thread auf die readylist zugreifen kann
    first = (Entrant*) readylist.dequeue();
    readylist_lock.unlock();
    if (first) {
      go(*first);
    }
    else {
      idle();
    }
  }
}

void Scheduler::ready (Entrant& that) {
  if( &that != active() ) {
readylist_lock.lock();
    readylist.enqueue (&that);
readylist_lock.unlock();
```

```
  }
}

void Scheduler::kill (Entrant& that) {
  readylist.remove (&that);
}

void Scheduler::idle() {
  if(lapic.get_processor_id() == 0) {
    do_idle = true;
  }
  else {
    do_idle2nd = true;
  }
  Entrant* idle = (Entrant*) idlelist.dequeue();
  dispatch (*idle);
}

void Scheduler::exit () {
  // hole naechsten Prozess aus ready-Liste.
  // wenn kein Prozess verfuegbar, gehe in idle-Loop
  readylist_lock.lock();
  Entrant* next = (Entrant*) readylist.dequeue();
  readylist_lock.unlock();
  if( next == 0 ) {
    idle();
  }
  else {
    dispatch (*next);
  }
}

void Scheduler::resume () {
  int processor_id = lapic.get_processor_id();
  if((!processor_id && !do_idle) || (processor_id && !do_idle2nd)) {
    Entrant* next;
    readylist_lock.lock();
    next = (Entrant*) readylist.dequeue ();
    readylist_lock.unlock();
    if (next) {
      readylist_lock.lock();
      readylist.enqueue ((Entrant*) active());
      readylist_lock.unlock();
      dispatch (*next);
    }
  }
}
```

dispatch.h:

```
/*****************************************************************************/
/* Betriebssysteme                                                           */
/*---------------------------------------------------------------------------*/
/*                                                                           */
/*                         D I S P A T C H E R                               */
/*                                                                           */
/*---------------------------------------------------------------------------*/
/* Implementierung des Dispatcher.                                           */
/* Der Dispatcher verwaltet den life-Pointer, der die jeweils aktive         */
/* Koroutine angibt. mit go() wird der life Pointer initialisiert und die    */
/* erste Koroutine gestartet, alle weiteren Kontextwechsel werden mit        */
/* dispatch() ausgeloest. active() liefert den life Pointer zurueck.         */
/*****************************************************************************/

#ifndef __dispatch_include__
#define __dispatch_include__

#include "thread/coroutine.h"
```

```
class Dispatcher
 {
private:
  Coroutine* life;
  Coroutine* life2nd;
public:
  Dispatcher () : life (0), life2nd(0) {}
  void go (Coroutine& first);
  void dispatch (Coroutine& next);
  Coroutine* active ();
 };

#endif
```

dispatch.cc:

```
/*****************************************************************************/
/* Betriebssysteme                                                           */
/*---------------------------------------------------------------------------*/
/*                                                                           */
/*                         D I S P A T C H E R                               */
/*                                                                           */
/*---------------------------------------------------------------------------*/
/* Implementierung des Dispatcher.                                           */
/* Der Dispatcher verwaltet den life-Pointer, der die jeweils aktive         */
/* Koroutine angibt. mit go() wird der life Pointer initialisiert und die    */
/* erste Koroutine gestartet, alle weiteren Kontextwechsel werden mit        */
/* dispatch() ausgeloest. active() liefert den life Pointer zurueck.         */
/*****************************************************************************/

#include "thread/dispatch.h"
#include "smp/lapic.h"
#include "device/cgastr.h"

extern LAPIC lapic;
extern CGA_Stream kout;

void Dispatcher::go (Coroutine& first) {
  if(lapic.get_processor_id() == 0) {
    if (!life) {
      life = &first;
      first.go();
    }
  }
  else {
    if (!life2nd) {
      life2nd = &first;
      first.go();
    }
  }
}

void Dispatcher::dispatch (Coroutine& next) {
  if(lapic.get_processor_id() == 0) {
    Coroutine* current = life;
    life = &next;
    current->resume (next);
  }
  else {
    Coroutine* current = life2nd;
    life2nd = &next;
    current->resume (next);
  }
}

Coroutine* Dispatcher::active() {
  if(lapic.get_processor_id() == 0) {
    return life;
  }
```

```
  else {
    return life2nd;
  }
}
```

guardian.cc:

```
/*****************************************************************************/
/* Betriebssysteme                                                           */
/*---------------------------------------------------------------------------*/
/*                                                                           */
/*                            G U A R D I A N                                */
/*                                                                           */
/*---------------------------------------------------------------------------*/
/* Zentrale Unterbrechungsbehandlungsroutine des Systems.                    */
/* Der Parameter gibt die Nummer des aufgetretenen Interrupts an.            */
/*****************************************************************************/

/* INCLUDES */

#include "machine/plugbox.h"
#include "smp/lapic.h"

extern Plugbox plugbox;
extern LAPIC lapic;

#include "guard/guard.h"

extern Guard guard;

#include "device/cgastr.h"

extern CGA_Stream kout;

/* FUNKTIONEN */

extern "C" void guardian (unsigned int slot);

/* GUARDIAN: Low-Level Interrupt-Behandlung. Die Funktion wird spaeter noch */
/*           erweitert.                                                      */

void guardian (unsigned int slot) {
  bool help;

  Gate* gate;

  gate = &(plugbox.report (slot));

  help = gate->prologue(slot);
  lapic.ack_APIC_irq();
  if(help) {
    guard.relay (gate);
  }
 }
```

locker.h:

```
/******************************************************************************/
/* Betriebssysteme                                                            */
/*----------------------------------------------------------------------------*/
/*                                                                            */
/*                              L O C K E R                                   */
/*                                                                            */
/*----------------------------------------------------------------------------*/
/* Die Klasse Locker implementiert eine Sperrvariable, die verwendet wird,    */
/* um kritische Abschnitte zu schuetzen. Die Variable zeigt allerdings nur    */
/* an, ob der kritische Abschnitt frei ist. Ein eventuelles Warten und der    */
/* Schutz der fuer diese Klasse notwendigen Zaehlfunktion muss ausserhalb     */
/* erfolgen.                                                                  */
/******************************************************************************/

#ifndef __Locker_include__
#define __Locker_include__

#include "device/cgastr.h"
#include "machine/spinlock.h"

extern CGA_Stream kout;

class Locker
 {
protected:
  Spinlock guard_lock;
public:
  // LOCKER : Konstruktor zur Initialisierung der Sperrvariablen (frei)
  Locker () {};

  // ENTER: Diese Methode muss aufgerufen werden, wenn der kritische
  //        Abschnitt betreten wird.
  void enter () {
    guard_lock.lock();
  };

  // RETNE: Der kritische Abschnitt wird verlassen.
  void retne () {
    guard_lock.unlock();
  };

  // AVAIL: Zeigt an, ob der kritische Abschnitt frei ist.
  int avail () {
    return (guard_lock.get_status() == 0);
  };

};

#endif
```

main.cc:

```
/******************************************************************************/
/* Betriebssysteme                                                            */
/*----------------------------------------------------------------------------*/
/*                                                                            */
/*                              M A I N                                       */
/*                                                                            */
/*----------------------------------------------------------------------------*/
/* Nachdem der 1.Prozessor vom Real-mode in den Protected-Mode gewechselt     */
/* springt er nach main.cc                                                    */
/******************************************************************************/

#include "user/appl3.h"
#include "user/idle.h"
```

```
#include "device/cgastr.h"

CGA_Stream kout;

#include "machine/pic.h"

PIC pic;

#include "machine/cpu.h"

CPU cpu;

#include "machine/plugbox.h"

Plugbox plugbox;

#include "machine/spinlock.h"

Spinlock kout_lock;
Spinlock epilog_lock;
Spinlock readylist_lock;

#include "guard/secure.h"


#include "guard/guard.h"

Guard guard;

#include "device/watch.h"

Watch watch(20000);

#include "syscall/guarded_keyboard.h"

Guarded_Keyboard keyboard;

#include "syscall/guarded_organizer.h"
#include "syscall/thread.h"

Guarded_Organizer organizer;

#include "meeting/bellringer.h"

Bellringer bellringer;

#include "smp/lapic.h"

LAPIC lapic;

#include "smp/io_apic.h"

IO_APIC io_apic;

#include "syscall/guarded_semaphore.h"

Guarded_Semaphore sema_kout(1);

#include "smp/smp_detect.h"

#include "smp/smp_activate.h"

int memcpy(unsigned char* source, unsigned char* dest, int n);
void aus_fps_zu_lesende_daten_setzen();
// Die folgende Funktion wird in setup2nd.asm implementiert.
extern "C" void setup2nd();

// werden benötigt um sie für den simulator zu setzen, der sie nicht aus der fps einlesen kann
extern unsigned int num_processors_present; /* Anzahl von Prozessoren */
extern unsigned char present_cpu_ID[MAX_NR_CPUS]; /* the CPUs local APIC ID of all CPUs */
```

```
extern int lapic_version[MAX_NR_CPUS];   // Version des lokalen APIC

static unsigned long idle_stack[1024];
static unsigned long idle2_stack[1024];

static unsigned long appl3_stack[4096];

int main() {

  kout << endl;

  // der setup code für den zweiten prozessor wird in den speicher kopiert, an die stelle, an der
  // der prozessor, nachdem der aufgeweckt wurde, mit der ausführung von befehlen beginnt
  unsigned char* trampoline = (unsigned char*)0x00001000;
  memcpy((unsigned char*)setup2nd,trampoline,1024);

  if(smp_detect()) {
    kout << "This is a MultiProcessorSystem =)" << endl;
    smp_init();
  }

  // nur nötig da der emulator die daten nicht aus der fps lesen kann, da sie nicht vorhanden ist
//aus_fps_zu_lesende_daten_setzen();
//smp_init();

  for(int i = 0;i < 100000;i++) {
    for(int j = 0;j < 1;j++) {
    }
  }

  // Erzeugen und anmelden von Idle-Threads für jeden Prozessor
  Idle idle(idle_stack+sizeof(idle_stack));
  Idle idle2(idle2_stack+sizeof(idle2_stack));
  organizer.idlelist.enqueue(&idle);
  organizer.idlelist.enqueue(&idle2);

  // Erzeugen und anmelden der Pacman-Anwendung
  Application3 application3(appl3_stack+sizeof(appl3_stack));

  int x, y;
  // Bildschirm loeschen
  for (y=0; y<25; y++)
    for (x=0; x<80; x++)
      kout.show (x, y, ' ', CGA_Screen::STD_ATTR);

  // Startmeldung ausgeben
  kout.setpos (36,10);
  kout << "MP-Stubs" << endl << endl << dec;

  // MAIN ist Teil des Kerns, daher muessen Interrupt-Aktivitaeten
  // noch verzoegert werden. Der erste aktivierte Thread (Application)
  // wird die Sperre in kickoff.cc wieder freigeben.
  guard.enter ();

  // Tastatur einstöpseln
  keyboard.Keyboard::plugin();

  // Uhr aufziehen
  watch.windup ();

  // Anwendung im Scheduler anmelden
  organizer.Organizer::ready(application3);

  // Scheduler starten
  organizer.Organizer::schedule ();

  while(true){
  }
  return 0;
}
```

```
// diese Funktion wird für den emulator aufgerufen, es werden alle informationen, die normalerweise
// aus der fps gelesen werden gesetzt, da die fps im emulator nicht implementiert ist
void aus_fps_zu_lesende_daten_setzen() {
  present_cpu_ID[0] = 0;
  present_cpu_ID[1] = 1;
  num_processors_present = 2;
  lapic_version[0]= ~0;
  lapic_version[1]= ~0;
}


int memcpy(unsigned char* source, unsigned char* dest, int n) {
  unsigned char* source_hlp = source;
  unsigned char* dest_hlp = dest;
  for ( ; n-- ; source_hlp++, dest_hlp++) {
    *dest_hlp = *source_hlp;
  }
  return 0;
}
```

main2nd.cc:

```
/******************************************************************************/
/* Betriebssysteme                                                            */
/*----------------------------------------------------------------------------*/
/*                                                                            */
/*                            M A I N  2 N D                                  */
/*                                                                            */
/*----------------------------------------------------------------------------*/
/* Nachdem der 2.Prozessor vom Real-mode in den Protected-Mode gewechselt     */
/* springt er nach main2nd.cc                                                 */
/******************************************************************************/

/* INCLUDES */
#include "smp/smp_activate.h"
#include "syscall/guarded_organizer.h"

extern "C" void main2nd();

/* GLOBALE VARIABLEN */
extern Guarded_Organizer organizer;

void main2nd() {

  // nachdem der zweite prozessor aufgeweckt wurde, meldet er dem ersten, dass er sich
  // konfiguriert hat und nun bereit ist
  smp_callin();

  // es wird gewartet, bis der erste prozessor den scheduler gestartet und eine anwendung
  // angemeldet hat
  for(int i = 0;i < 100000;i++) {
    for(int j = 0;j < 20;j++) {
    }
  }

  // dann beginnt der zweite prozessor ebenfalls mit der arbeit
  organizer.schedule();

  while(true) {}

}
```

smp.h:

```
/*****************************************************************************/
/* Betriebssysteme                                                           */
/*---------------------------------------------------------------------------*/
/*                                                                           */
/*                                S M P                                      */
/*                                                                           */
/*---------------------------------------------------------------------------*/
/* In smp.h werden Konstanten definiert die in einem Multiprozessorsystem    */
/* benötigt werden                                                           */
/*****************************************************************************/

#ifndef __smp_include__
#define __smp_include__

#define MAX_NR_CPUS      15

#endif
```

Literaturverzeichnis

[1] Intel MultiProcessor Specification Version 1.4; 1997

[2] Intel 64 and IA-32 Architectures Software Developer's Manual Volume 2A: Instruction Set Reference, A-M; 2007

[3] Intel 64 and IA-32 Architectures Software Developer's Manual Volume 2B: Instruction Set Reference, N-Z; 2007

[4] Intel 64 and IA-32 Architectures Software Developer's Manual Volume 3A: System Programming Guide, Part 1; 2007

[5] Intel Pentium Processor Family Developer's Manual Volume 3: Architecture and Programming Manual; 1995

[6] Intel 82093AA I/O Advanced Programmable Interrupt Controller (IOAPIC); 1996

[7] Robert Love. Linux Kernel Development Second Edition; 2005

[8] Vorlesungsskript Betriebssysteme, Friedrich-Alexander-Universität Erlangen-Nürnberg, Lehrstuhl für Informatik 4; WS 2007/08

[9] Vorlesungsskript Cluster-Computing, Friedrich-Alexander-Universität Erlangen-Nürnberg, Lehrstuhl für Informatik 2; SS 2007

[10] Dominik Moser. TopsySMP - A Small Multi-Threaded Micorkernel for Symmetrical Multiprocessing Hardware Architectures. Diploma Thesis, Swiss Federal Institute of Technology Zurich; 1999

Abbildungsverzeichnis

Tabellenverzeichnis