

单位代码: 10293 密 级:

南京邮电大学

专 业 学 位 硕 士 论 文



论文题目: Intel Xeon-E5 多处理器的
UEFI 固件驱动开发

学 号 1210012424

姓 名 张 孝 亭

导 师 邱 晓 晖

专业学位类别 工 程 硕 士

类 型 全 日 制

专业（领域） 电子与通信工程

论文提交日期 2013 年 3 月

The Firmware Driver Development of Intel Xeon-E5 Multiprocessor in UEFI

Thesis Submitted to Nanjing University of Posts and
Telecommunications for the Degree of
Master of Engineering



By

Zhang Xiaoting

Supervisor: Prof. Qiu Xiaohui

March 2013

南京邮电大学学位论文原创性声明

本人声明所呈交的学位论文是我个人在导师指导下进行的研究工作及取得的研究成果。尽我所知，除了文中特别加以标注和致谢的地方外，论文中不包含其他人已经发表或撰写过的研究成果，也不包含为获得南京邮电大学或其它教育机构的学位或证书而使用过的材料。与我一同工作的同志对本研究所做的任何贡献均已在论文中作了明确的说明并表示了谢意。

本人学位论文及涉及相关资料若有不实，愿意承担一切相关的法律责任。

研究生签名：_____ 日期：_____

南京邮电大学学位论文使用授权声明

本人授权南京邮电大学可以保留并向国家有关部门或机构送交论文的复印件和电子文档；允许论文被查阅和借阅；可以将学位论文的全部或部分内容编入有关数据库进行检索；可以采用影印、缩印或扫描等复制手段保存、汇编本学位论文。本文电子文档的内容和纸质论文的内容相一致。论文的公布（包括刊登）授权南京邮电大学研究生院办理。

涉密学位论文在解密后适用本授权书。

研究生签名：_____ 导师签名：_____ 日期：_____

摘要

UEFI (Unified Extensible Firmware Interface) 是下一代计算机固件接口标准, 旨在替代传统的 BIOS, Intel 最新发布的基于 UEFI 标准的开源项目是 EDKII (EFI Development Kit) 框架。随着 UEFI 的进一步推广和开源化, 越来越多的 BIOS 厂商和研发机构参与了 UEFI 的研究和开发。然而, 受限于知识产权的保护, 有关 UEFI 框架下 Intel IA32 and 64 体系架构的多处理器的研究和开发工作仍然主要由 Intel 工程师完成, 而且在 EDKII 框架中多处理器固件驱动是作为程序二进制文件存在的, 这就造成了有关 IA32 and 64 多处理器研究方面的局限性, 对于多处理器固件驱动开发也没有系统的指导。

本论文所研究的就是基于 Intel IA32 and 64 体系架构的 Intel Xeon-E5 多处理器各个功能模块的实现机制, 并基于此实现 Xeon-E5 多处理器的 UEFI 固件驱动开发。现代软件工程中驱动开发项目一般都要遵循相关的开发说明书, 本课题的开发过程也大量参考了 IA32 and 64 体系架构的软件开发手册和规范说明书, 在具体的程序开发中采用模块化的程序设计方法对要实现的各个功能分别设计和实现。论文中重点描述了各个功能所管理和使用的数据结构和寄存器, 并研究和分析了各个功能模块具体的实现机制, 最后用伪代码实例的形式阐述如何用代码实现多处理器的各项功能。

本项目开发所使用的是配置 Intel Xeon-E5 双处理器的最新服务器平台, 目前基于 Sandy Bridge 核心的 Xeon 处理器是服务器平台所采用的最新的处理器, 遵循 IA32 and 64 体系架构。本论文中研究和实现了 IA32 and 64 体系架构中定义的几乎全部的功能, 除此之外, 论文中创新性的对多处理器的通信和管理机制的作了详细的研究并给出了伪代码实例, 为多处理器机制的开发提供了一定的参考价值。

关键词: UEFI, IA32 and 64 体系架构, 内存管理, 中断和异常管理, 多任务管理, 多处理器机制

Abstract

UEFI(Unified Extensible Firmware Interface) is the next generation firmware interface standard, aimed at replacing the traditional BIOS. The latest open source project based on UEFI standard released by Intel is EDKII (EFI Dev Kit) framework. With the further promotion and open sourcing of UEFI, more and more BIOS manufacturers and research institutes get involved in UEFI research and development. Limited to the protection of intellectual property, however, research and development of multiprocessor under Intel IA32 and 64 architecture is still mainly accomplished by Intel engineers. And firmware driver of multiprocessor exists as binary file in EDKII framework. These two points cause limitation of IA32 and 64 multiprocessor research, at the same time, there is no systematic guidance for multiprocessor firmware and driver development.

This paper studies each functional module's realization mechanism of the Intel Xeon-E5 multiprocessor under IA32 and 64 architecture, and realizes Xeon-E5 multiprocessor UEFI firmware driver development based on above study. For modern software engineering, driver development project generally follows related development specifications, this subject also largely references software development manual and standard specification of IA32 and 64 architecture. In addition, specific program development uses modular programming methods to realize each function design and implementation. The thesis describes the data structures and registers used and managed by each functional module, as well as analysis of each functional realization mechanism. At last, it uses pseudo code to state how to implement each function of the processor.

The development platform in this project is the latest server platform configured with Intel Xeon-E5 double processor. And at present, Xeon processor based on Sandy Bridge core is the most advanced processor on server platform, that follows IA32 and 64 architecture. This thesis researches and realizes almost all of processor functions defined by IA32 and 64 architecture. Beyond that, the thesis innovatively researches multiprocessor communication and management mechanism in detail and gives the pseudo code examples, which provides the development of multiprocessor system with a certain reference value.

Key words: UEFI, IA32 and 64 architecture, Memory Management, Exception and Interrupt Management, Multi-task Management, Multiprocessor Management

目录

专用术语注释表.....	1
第一章 引言.....	2
1.1 UEFI 概况和技术特点.....	2
1.1.1 UEFI 国内外研究概况和研究趋势.....	2
1.1.2 UEFI 的局限性和安全隐患.....	3
1.2 UEFI 接口架构和各阶段分析.....	4
1.2.1 UEFI 架构组成.....	4
1.2.2 UEFI 各阶段的分析.....	5
1.3 选题背景、研究意义及先进性.....	7
1.3.1 选题背景.....	7
1.3.2 研究意义及先进性.....	7
1.4 论文研究内容.....	8
第二章 固件驱动总体设计.....	10
2.1 固件驱动开发总体实现.....	10
2.1.1 系统级体系结构.....	10
2.1.2 驱动开发总体介绍.....	11
2.2 固件驱动各模块概述.....	12
2.2.1 内存管理.....	12
2.2.2 中断和异常管理.....	13
2.2.3 多任务管理.....	14
2.2.4 多处理器管理.....	14
2.3 开发环境及开发流程.....	15
2.3.1 EDKII 开发框架.....	15
2.3.2 开发调试过程.....	15
2.4 项目实现目标.....	17
2.5 本章小结.....	18
第三章 内存管理.....	19
3.1 内存管理概述.....	19
3.2 系统分段机制模型.....	20
3.2.1 基本的内存平面模型.....	20
3.2.2 多重分段模型.....	21
3.3 分段机制.....	22
3.3.1 段选择子和段寄存器.....	22
3.3.2 段描述符和段描述符表.....	22
3.3.3 逻辑地址到线性地址的转换.....	24
3.3.4 分段机制的算法实现.....	25
3.4 分页机制.....	28
3.4.1 分页模式以及模式切换.....	28
3.4.2 分页模式控制比特位.....	29
3.4.3 分页机制原理：分层级分页.....	29
3.4.4 32-bit 分页以及算法实现.....	31
3.5 本章小结.....	35
第四章 中断和异常管理.....	37
4.1 中断和异常管理概述.....	37

4.2 中断和异常向量.....	38
4.3 异常源.....	38
4.3.1 程序错误异常.....	38
4.3.2 软件产生的异常.....	38
4.3.3 机器检查异常.....	39
4.4 异常分类.....	39
4.5 中断源.....	39
4.5.1 外部中断.....	39
4.5.2 软件产生的中断.....	40
4.6 同时发生的异常和中断的优先级.....	41
4.7 程序或任务的继续执行.....	41
4.8 中断描述符表和中断描述符.....	42
4.8.1 中断描述符表.....	42
4.8.2 IDT 描述符.....	43
4.9 异常和中断处理机制.....	44
4.9.1 异常或中断处理程序调用过程.....	45
4.9.2 中断任务的执行.....	46
4.9.3 中断或异常处理机制的算法实现.....	47
4.10 本章小结.....	49
第五章 任务管理.....	51
5.1 任务管理综述.....	51
5.1.1 任务结构.....	52
5.1.2 任务的状态信息.....	52
5.2 调度任务的执行.....	53
5.3 用于任务管理的数据结构.....	54
5.3.1 任务状态段.....	54
5.3.2 TSS 描述符.....	56
5.3.3 任务寄存器.....	57
5.3.4 任务通道描述符.....	58
5.4 任务切换机制及算法实现.....	58
5.4.1 任务切换机制.....	59
5.4.2 任务切换机制实现.....	60
5.5 本章小结.....	61
第六章 多处理器机制.....	62
6.1 上锁的原子操作.....	62
6.1.1 给总线上锁.....	63
6.2 内存有序化.....	63
6.3 多处理器的初始化.....	64
6.3.1 BSP 和 AP.....	65
6.3.2 Intel Xeon 处理器的 MP 初始化协议算法.....	65
6.3.3 BSP 初始化工作.....	66
6.3.4 AP 初始化工作.....	68
6.4 多处理器的管理.....	68
6.4.1 BSP 或 AP 的确定.....	69
6.4.2 逻辑处理器的信息.....	70
6.4.3 BSP 和 AP 的切换.....	73

6.4.4 启动 AP 执行程序..... 74

6.5 本章小结..... 76

第七章 总结与展望..... 77

7.1 课题总结..... 77

7.2 研究展望..... 78

参考文献..... 79

附录 1 攻读硕士学位期间参加的科研项目..... 81

致谢..... 82

专用术语注释表

缩略词说明:

UEFI	Unified Extensible Firmware Interface	统一可扩展固件接口
EFI	Extensible Firmware Interface	可扩展固件接口
BIOS	Basic Input and Output System	基本输入输出系统
SEC	Security Core	安全核心阶段
PEI	Pre-EFI Initialization	预 EFI 初始化阶段
DXE	Driver Execution Environment	驱动执行环境
BDS	Boot Device Selection	启动设备选择阶段
IA	Intel Architecture	Intel 处理器架构
EDK	EFI Development Kit	EFI 开发框架
GDT	Global Description Table	全局描述符表
LDT	Local Description Table	本地描述符表
IDT	Interrupt Description Table	中断描述符表
TSS	Task State Segment	任务状态段
DPL	Description Privilege Level	描述符权限级
RPL	Requested Privilege Level	请求的权限级
CPL	Current Privilege Level	当前权限级
APIC	Advanced Programmable Interrupt Controller	高级可编程中断控制器
NMI	Non-maskable Interrupt	不可屏蔽中断
PDE	Page Directory Entry	页路径入口
PTE	Page Table Entry	页表入口
BSP	Bootstrap Processor	引导处理器
AP	Application Processor	应用处理器
MP	Multiprocessor	多处理器
ACPI	Advanced Configuration and Power Interface	高级配置与电源接口
GUID	Globally Unique Identifier	全局唯一标识符
BIST	Build-In Self-Test	内建自测试
PAE	Physical Address Extension	物理地址扩展
32e	32bit mode Extension	32bits 扩展模式
IPI	Inter-processor Interrupt	处理器间中断信号

第一章 引言

作为全球最大的 CPU 芯片研发和生产厂商，Intel 在 2000 年推出了 BIOS 新一代标准接口 UEFI，板级初始化的所有模块都被包含在 UEFI 中，这其中包括 CPU 驱动程序。本章首先介绍 UEFI 架构的概况以及 UEFI 各阶段的分析，然后介绍本论文的选题背景，意义和先进性，最后介绍本论文要研究的内容。

1.1 UEFI 概况和技术特点

1.1.1 UEFI 国内外研究概况和研究趋势

作为连接操作系统与硬件体系之间的桥梁，传统 BIOS 为 PC 的发展做出了重要贡献。BIOS 作为底层硬件和软件之间的接口，负责执行软件对硬件的即时操作要求，它主要负责操作系统执行前的初始化工作，包括检查系统配置和连接计算机内各种不同的硬件和操作系统^[1]。作为由低级汇编语言写成的软件，BIOS 以 16 位汇编代码、寄存器参数调用方式、静态链接，以及 1MB 以下内存固定编址的形式存在了很长一段时间。正是这一陈旧的运行方式，迫使 Intel、AMD 在开发新款 CPU 时，必须考虑加入导致系统性能大大降低的兼容模式。用户操作体验不佳、代码编写复杂等 BIOS 的缺点也导致设计者怨声载道^[2]。

2000 年，Intel 向业界推出了 BIOS 的新一代标准接口 EFI，并在当时的安腾服务器平台上采用 EFI 技术。传统的 BIOS 采用汇编语言编写，越来越难以解决硬件不断更新换代对软件同步更新的需要，而新的 EFI 采用高度模块化、动态链接和 C 语言风格的参数堆栈传递方式的形式构建系统，比传统的 BIOS 更具扩展性和易于实现性。另外，EFI 驱动程序可以不由运行在 CPU 上的代码组成，而是由 EFI 字节代码编写而成，保证了在不同 CPU 架构上的兼容性。Intel 将 EFI 定义为一个可扩展的、标准化的固件接口规范，不同于传统 BIOS 的固定的、缺乏文档的、完全基于经验和约定的一个事实标准。

从核心来看，EFI 很像一个被简化的操作系统，介于硬件设备和高级操作系统（如 Windows 系统）之间^[1]。EFI 内置了图像驱动功能，能够提供一个高分辨率的彩色图形环境，并且支持鼠标点击操作，明显有别于传统 BIOS 单调的纯文本界面，如图 1.1。

与传统 BIOS 的另一显著不同点是，EFI 使用全球最广泛的高级语言 C 语言进行编写，摆脱了传统 BIOS 复杂的 16 位汇编语言代码编写方式。这意味着有更多工程师可参与 EFI 的

开发工作，有利于平台创新快速发展。目前，EFI 的应用已由服务器领域扩展至 PC 领域，苹果在其 x86 PC 机上已采用了 EFI^[1]。与此同时，EFI 技术向消费电子、家用设备领域的延伸也从未停止。例如，通过 EFI 技术，当计算机未进入操作系统前，就可接入互联网。

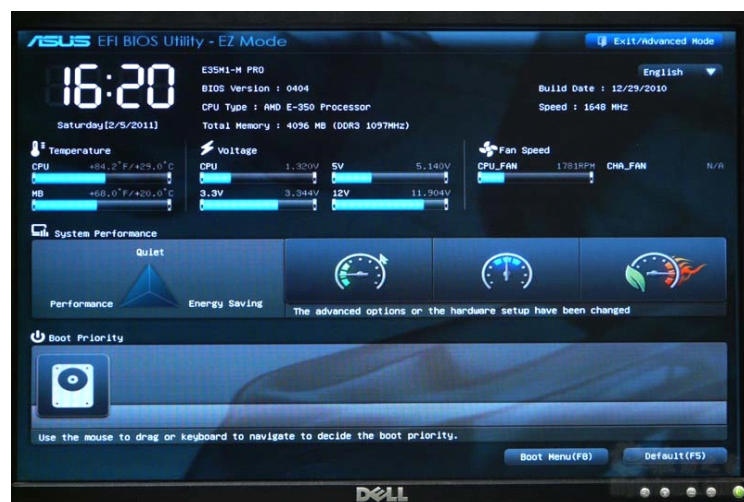


图 1.1 华硕 UEFI BIOS 界面

EFI 在安腾服务器平台上的应用和它的前景，让更多人看到了 EFI 的魅力。2005 年，在工业界达成共识的基础上，Intel 主导成立了 UEFI 联盟^[3]。UEFI 联盟是由 Intel、微软、惠普等厂商于 2005 年共同发起成立的国际组织。该组织约定定期召开 UEFI 技术大会。目前联盟已有 86 家企业成员，这些企业成员共分为三级：推广者、贡献者和接受者，这三类成员在联盟中分别承担不同的权责和义务。UEFI 联盟的工作小组包括规范工作组（USWG）、测试工作组（UTWG）、平台初始化工作组（PIWG）和业界联络工作组（ICWG），四个工作组通过对行业进行大量、多样化的教育和推广，促使业界尽快认识和采用 UEFI 标准^[4]。

目前，该联盟发布了两个最新规范：UEFI 2.3 和新的平台初始化规范 PI1.2。前者是用来定义系统固件与操作系统或其它高级软件（包括固件驱动程序）之间的接口，后者用来保证由不同企业（如芯片厂商、固件开发商和维护固件代码的组织等）提供的固件组件之间的互操作性。

在美国成功举办过两次 UEFI 技术大会后，2007 年，UEFI 技术大会首度移师中国南京。而该联盟常务执行副总裁魏东的宣讲，也让国内众多 IT 厂商更全面地了解了 UEFI 联盟及他们的职责。

1.1.2 UEFI 的局限性和安全隐患

凡事都有两面性，UEFI 技术在为我们提供模块化、结构层次清楚、网络应用、高级语言开发、易于实现等优点的同时，也带来了一些安全隐患，比如：

(1) 由于 UEFI 在 DXE 阶段就引入了完整的网络应用, 比如 TCP、UDP、MTFTP 等支持, 这一方面为我们提供了调试网络设备的便利, 同时也为下一步的网络安全留下了隐患。

(2) UEFI 良好的扩展性使其更多的内部接口被暴露出来, 同时其核心代码部分开源使得黑客无需反汇编代码, 仅通过阅读核心源代码的方法就有可能找出其中的漏洞和缺陷。

(3) 由于 UEFI 采用高级语言开发, 相比之前的汇编语言, 降低了开发门槛, 同时提供了丰富的调试接口, 尤其是 DXE 阶段 UEFI Shell 这样的调试接口, 非常有可能引起新一轮针对 BIOS SMM (System Management Mode) 的攻击^{[3][5]}。

(4) UEFI 作为一种新型的技术, 其阶段划分复杂, 每个阶段都有一个单独的控制核心, 比如 PEI 与 DXE 阶段均由相应的分发器负责。对于简单计算机平台, 尤其是对于嵌入式设备而言, 如此多阶段与阶段之间的衔接必然会在启动速度上不如传统的 Bootloader。

1.2 UEFI 接口架构和各阶段分析

1.2.1 UEFI 架构组成

EFI 定义了操作系统和平台固件之间的接口规范, 它描述了一个对平台的可编程的接口, 这个平台包括 CPU, 主板, 芯片组和其他组成部分^{[3][6]}。

UEFI 主要包括软件架构和工业标准协议构件, 其中软件架构是主要部分。包括基于 EFI 固件管理的对象, EFI 系统数据表, EFI 可执行文件, EFI 服务, Handle 数据库。

(1) 基于 EFI 固件管理的对象—包括 I/O 设备, 内存和事件, 通过这些对象可以获得系统的状态。

(2) EFI 系统数据表—用来与系统交互所用到的主要的数据信息表, 它的指针作为函数参数被传递到每个应用程序和驱动中。从这个数据结构, EFI 应用程序就可以得到系统的配置信息和多个 EFI 服务程序。

(3) EFI 可执行文件—包括各种 EFI 应用程序和驱动模块。驱动模块分为启动时驱动和运行时驱动。

(4) EFI 服务—包括 EFI 启动时服务, EFI 运行时服务, 和 protocol 服务, EFI 应用程序或驱动程序可以通过 EFI 系统数据表使用这些服务。

(5) Handle 数据表—由 handles 和 protocols 组成。在 EFI 初始化中, EFI 应用程序和 EFI 驱动会创建 handles, 并通过在 handle 上挂载 protocol 实现自身的功能^[3]。

除了软件架构, UEFI 还定义了一些工业界的标准协议构件, 如 ACPI 和 SMBIOS 等。图

1.2 描述了 UEFI 架构的各个模块、接口和标准协议之间的关系^[3]。

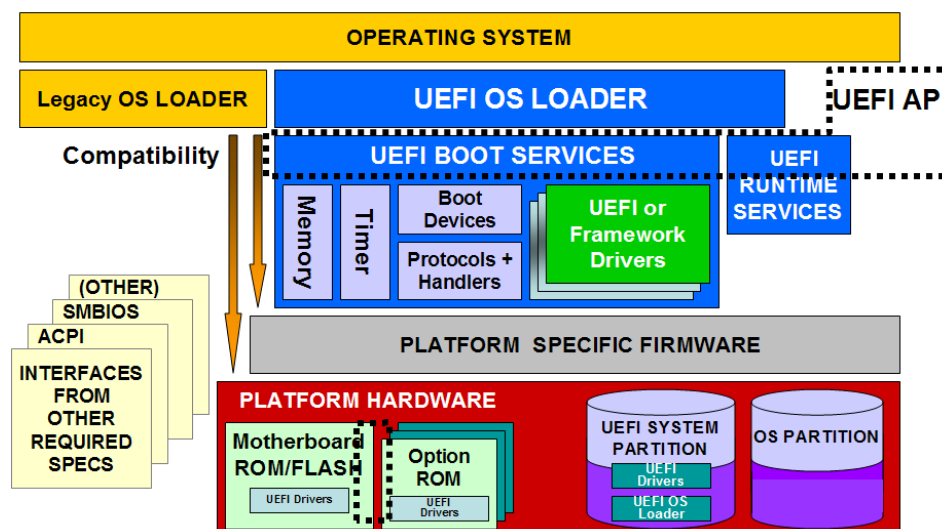


图 1.2 UEFI 架构模型

UEFI BIOS 主要采用面向对象的思想设计，逐层向上提供服务，在不同服务层的衔接方面采用统一化标准的接口。比如在 UEFI 启动服务 (BOOT SERVICES) 中，包含了内存管理、计时时钟、启动服务、启动协议和固件驱动等诸多对象，但是在对外的接口上重新对这些对象的接口进行封装，为 UEFI 操作系统加载层 (OS LOADER) 提供统一的接口。用户只需按照标准协议手册查找使用这些标准化的服务，而不需要关注到底是哪个对象提供的这种服务。

1.2.2 UEFI 各阶段的分析

作为 UEFI 联盟的发起者，Intel 长期以来都有专门的 Tiano 组负责 UEFI BIOS 的研究和开发。按照 UEFI 2.3.1 规范的定义，基于现代软件体系设计的思想，Tiano 按照 BIOS 启动过程的四个阶段来开发 UEFI BIOS: SEC, PEI, DXE, 和 BDS^{[3][7]}，其运行机理如图 1.3 所示。

SEC (Security Core) 是平台初始化的第一个阶段，是上电后最先执行的阶段。它需要处理所有的系统启动或重启的异常事件，例如系统冷启动；同时，它还会在主存储被初始化之前，构建出少量临时存储单元，用于系统在软件结构中所需的堆栈等。

因为 UEFI 选择了高级语言 C 作为 UEFI 的开发语言，所以这个阶段还要找到一块初始化过的内存，使 C 代码可以在给定的系统上执行。接下来的 PEI 和 DXE 阶段的划分也是基于这个原则。

PEI (Pre EFI) 是 UEFI 运行之前的初始化阶段，通常在 SEC 阶段之后立即执行。PEI 阶段会使用一些处理器的资源，比如 Cache 作为函数的调用栈，来派遣不同的 PEI 模块去做最基本的系统初始化以保证 DXE 阶段可以运行，主要包括初始化系统内存、描述系统内存和固件标签的 Flash 地址，并且把控制权交给 DXE 阶段。PEI 阶段也要考虑应急启动和休眠启动，

应急启动要求代码尺寸尽可能小，休眠启动要求代码运行尽可能快，所以 PEI 阶段的代码需要尽可能的简化。

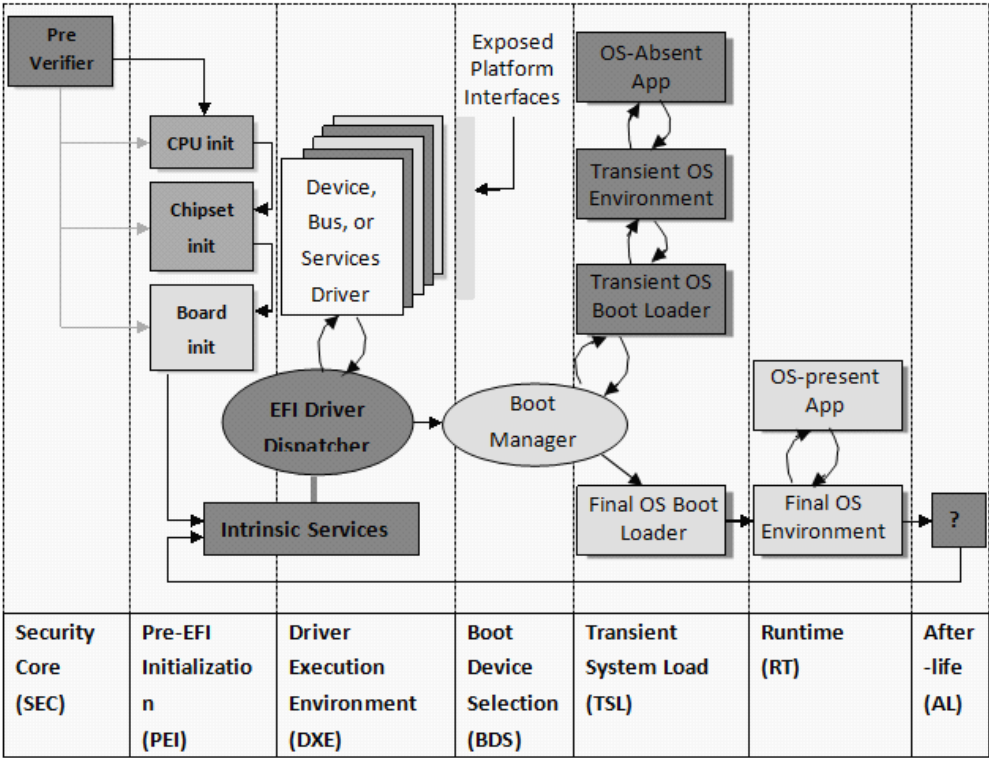


图 1.3 UEFI/Tiano 系统架构

PEI 到 DXE 阶段的转化是一个单向的过程，一旦为 DXE 阶段的初始化装入程序完成后，PEI 阶段代码将不可用，这时 DXE 成为一个设备齐全的操作环境。

系统的大多数初始化工作在 DXE（Driver Execution Environment）阶段中完成。在 DXE 阶段有几个组件，包括 DXE 核、DXE 派发程序和若干 DXE 驱动程序。DXE 核是 DXE 阶段的基础和核心模块，负责产生 UEFI 规范中定义的启动服务和运行时服务，并提供一组扩展的 DXE 服务。DXE 分派程序负责按照正确的顺序发现和运行 DXE 驱动程序。DXE 驱动程序负责初始化处理器，芯片组 and 平台组件，同时为系统服务、控制台设备和启动设备提供软件抽象，如本论文中的处理器驱动程序就是在 DXE 阶段被分配执行的。这些组件一起工作来初始化平台，并且提供启动一个操作系统所需的服务^[7]。

启动设备选择阶段 BDS（Boot Device Selection）与 DXE 阶段一起工作，创建一个控制台，并且尝试从可用的启动设备来启动操作系统。BDS 是控制权交给操作系统之前的最后一个阶段。在 BDS 阶段，可以向最终用户展示用户界面，也可以被 OEM 厂家用来修改定制特定的交互环境。

1.3 选题背景、研究意义及先进性

1.3.1 选题背景

Intel 作为 UEFI 联盟的发起者，其内部的 Tiano 组一直在进行 UEFI BIOS 的研究和开发。Tiano 组最新发布的 UEFI BIOS 的框架为 EDKII (EFI/Tiano Development Kit II) [8]。EDK 包括了一些由 Tiano 开发的 EFI 项目实例和相应的开发工具，Intel 已经在 BSD 许可证下公开了其框架的所有代码，以及一些 Intel 知识产权下的驱动程序二进制文件。此外，EDK 包含了一些驱动示例，可供设备驱动生产厂商参考。在 EDK 环境下，可以开发、调试以及测试 EFI 驱动、Option Rom 驱动和 EFI Shell 应用程序[9]。

EDKII 项目是针对 UEFI 社区的要求，对 UEFI BIOS 的开发框架进行管理，形成更好的项目开发和版本控制的环境[8][9]。EDK 的最初开发成员主要由 Intel 的平台工程师构成，但其开源的软件开发模式使得更多的 BIOS 生产厂商可以参与。这些生产厂商包括独立的硬件生产商、独立的 BIOS 生产商、OEM，也包括了操作系统开发机构和教育研究机构。

本论文所研究的 Intel Xeon-E5 多处理器 UEFI 固件驱动是作为 EDKII 开发项目的一部分进行的，本项目开发平台的硬件配置上采用两个基于 IA32 and 64 处理器架构的 Intel Xeon-E5 处理器，每个处理器均采用多核技术和超线程技术。处理器是系统中最重要的一部分，它的初始化以及功能的实现直接影响到整个系统的功能。多处理器固件驱动和其他 EFI 驱动一样也是在 DXE 阶段被分配执行的，它主要完成系统多处理器的初始化工作，使多处理器能够完成对系统内存的管理，中断和异常管理，多处理器管理等功能。

本论文中的处理器固件驱动开发的重点和难点在于开发平台采用了两个 Intel Xeon-E5 处理器，在其固件驱动中要完成多处理的管理和调度。本人有幸在 Intel 亚太研发中心进行了一年时间的实习，实习期间主要从事 EDKII 项目中多处理器固件驱动的调试工作，所以对本项目中的多处理器固件驱动的研发非常感兴趣。但是需要声明的是，由于要遵守 Intel 公司的保密协议，所以在论文中只使用伪代码实例代码源代码。

1.3.2 研究意义及先进性

随着 UEFI BIOS 在世界范围内的逐步推广，越来越多的 BIOS 生产厂商，操作系统研发机构以及教育机构开始参与 UEFI 方面的研究和开发。尽管这样，对于 UEFI BIOS 的处理器固件驱动开发仍然主要由 Intel 平台工程师完成，而且处理器驱动的源代码并不是开放的[6]。

同时随着处理器的更新换代，处理器性能和结构复杂性也不断升级，这对处理器驱动的开发人员来说是一个巨大的挑战。如何实现系统多处理器各功能的初始化，使得处理器稳定运行并且最新的处理器技术得以充分发挥是开发人员要解决的重点问题。本项目中开发平台采用 Intel 新发布的基于 Sandy Bridge 核心架构的 32nm 制成工艺 Xeon-E5 系列处理器，同时采用双处理器配置，这对以往 Nehalem 核心架构的单处理器具有一定的先进性。

Intel Xeon 系列处理器基于 IA32 and 64 架构，根据最新的 IA32 and 64 体系架构处理器开发手册的描述，在 EDKII 项目中处理器固件驱动的开发要完成内存管理，中断管理，任务管理以及多处理器管理等功能的初始化工作，为系统从 UEFI BIOS 过渡到操作系统准备好必要的条件，保证处理器在操作系统中正常而稳定的运行。同时在本项目中采用的是服务器平台，其处理器配置是两个 Intel Xeon-E5 处理器核，如何实现双处理器的管理以及处理器之间的调度也成为了多处理器驱动设计中重点解决的问题。通过对以上内容研究和实现，不仅完成了新一代 Xeon-E5 处理器各个管理功能的初始化，也为多处理器管理机制提供了重要的学习和参考依据，同时为 UEFI 标准的固件驱动的开发提供了重要的指导。

1.4 论文研究内容

本课题的研究内容主要包括以下几个方面：UEFI BIOS 的背景知识的介绍，多处理器固件驱动开发的总体设计以及各个模块的概述，根据总体设计详细研究各个功能的实现机制，并基于对实现机制的分析给出简要的代码实例。由于要遵守 Intel 公司的保密协议，本论文只使用了伪代码实例代码源代码。

本论文研究的重点有 4 个。其一是处理器内存管理机制的研究和实现。内存管理是对系统内存资源进行管理的机制，IA32 and 64 体系架构的内存管理分成两个部分：分段（segmentation）和分页（paging）。分段提供了将单独的代码，数据和堆栈隔离开的机制，这样多个任务可以互不干扰的运行在同一个处理器上。分页提供了实现传统分页需求的虚拟内存管理的机制，系统中程序执行环境的段按需的映射到物理内存中。论文将对分段和分页机制的原理作详细分析，并通过伪代码实现处理器内存管理功能。

其二是处理器中断和异常管理机制的研究和实现。中断和异常管理功能是处理器为了管理系统内外特殊情况的机制。它将结构化定义的中断或异常情况与中断向量号相对应，然后通过执行中断或异常处理程序实现中断或异常情况的响应。论文中将重点研究中中断和异常响应的过程，并通过具体实例介绍如何管理一个中断或异常情况。

其三是处理器任务管理机制的研究和实现。一个任务就是一个工作单元，此工作单元可

以用来执行一段程序，一个进程，一个操作系统服务，或者一个中断或异常处理程序等等。任务管理机制提供了如下功能：保存一个任务的状态信息，调度任务执行，和切换不同的任务。论文将重点研究 Intel Xeon 处理器任务管理功能的任务切换机制，并用伪代码片段描述任务切换机制的实现。

其四是处理器多处理器管理机制的研究和实现。当系统中的处理器是多个时，就需要对多处理器进行管理和调度。本论文将重点研究多处理器管理机制和处理器之间调度机制，并通过伪代码实现这些机制。

第二章 固件驱动总体设计

Intel Xeon 是基于 IA32 and 64 体系架构的处理器，该架构处理器为操作系统和系统开发软件提供了如下操作的支持^[10]：

- (1) 内存管理 (memory management)
- (2) 软件模块保护 (protection of software modules)
- (3) 多任务处理 (multitasking)
- (4) 异常和中断响应
- (5) 多处理器管理 (multiprocessing)
- (6) 高速缓存管理 (cache management)
- (7) 硬件资源和电源管理 (hardware resource and power management)

在 Xeon-E5 多处理器固件驱动开发过程中，我们需要参考 IA32 and 64 的开发手册^[10]完成对多处理器的初始化，使操作系统和系统软件稳定的执行上述操作。本章主要从总体上设计固件驱动开发要完成的工作，并对各个部分作简要的介绍，最后阐述开发需要的流程和方法，以及最后要实现的目标。

2.1 固件驱动开发总体实现

2.1.1 系统级体系结构

在作固件驱动总体设计时，需要了解处理器需要管理的系统资源，即系统级体系结构，处理器固件驱动的开发是运用系统资源的基础上完成的。系统级体系结构由一系列寄存器，数据结构和基本的系统级操作（如内存管理，异常和中断响应，任务管理，多处理器控制）的指令组成。

对于固件驱动要实现的具体模块，处理器所管理和使用的系统资源是不同的^{[10][11]}。

(1) 内存管理机制包含分页和分段两部分，段选择子 (segment selector) 和段描述符 (segment descriptor) 可以帮助处理器实现内存管理的功能。

(2) 对于多任务处理机制，处理器要依靠最重要的数据结构是任务状态段 (TSS)，其中定义了一个任务的执行环境，还有特殊的段选择子——通道 (gate)。

(3) 异常和中断管理用来处理来自系统内部或外部的中断和异常请求，这需要中断向量

号以及中段描述符表进行索引。

(4) 多处理器需要协调各个处理器之间调用的机制，它间接地使用这些资源。

(5) 对于高速缓存管理和电源管理，处理器需要管理很多的系统寄存器，如 EFLAGS 等。

图 2.1 是系统级体系结构的功能框图。

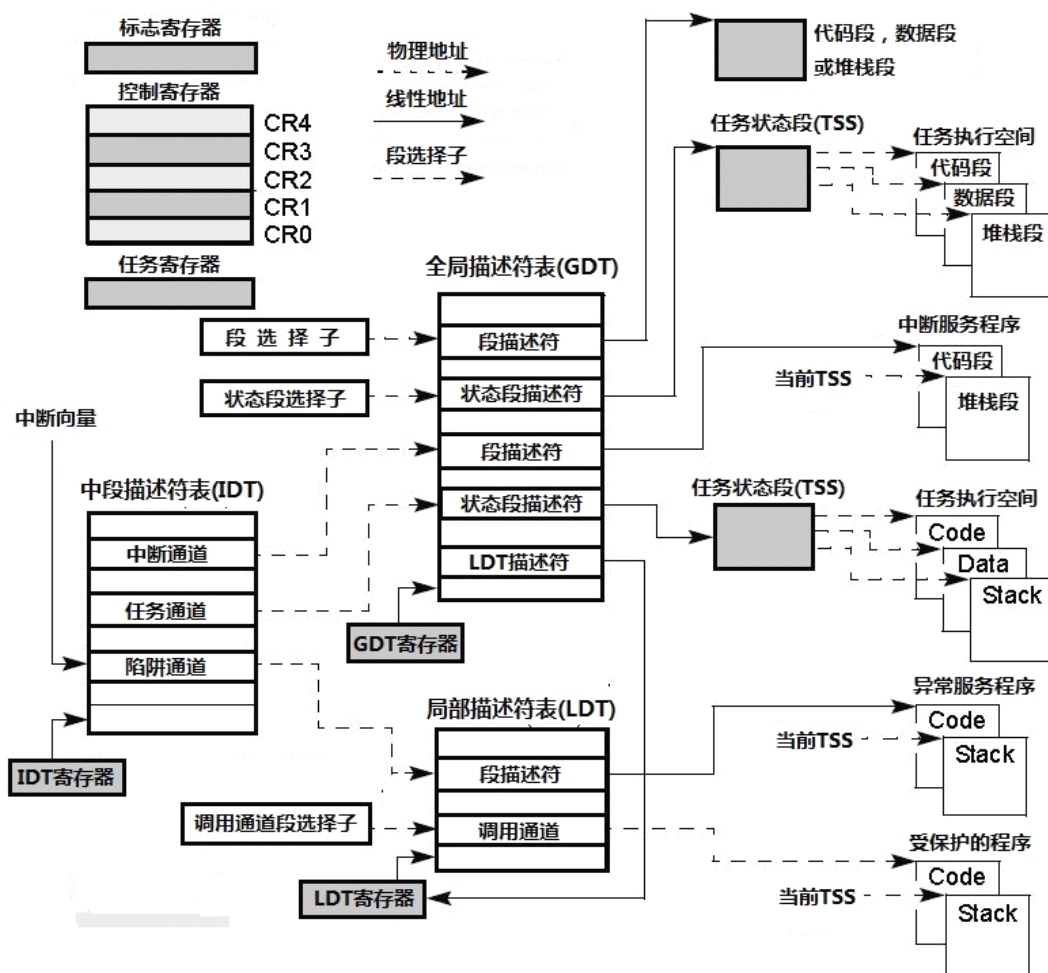


图 2.1 系统体系结构功能框图

2.1.2 驱动开发总体介绍

固件驱动是在运用上面系统体系结构的基础上，管理好系统资源，完成 IA32 and 64 架构要实现的操作系统操作的支持。本论文中对多处理器固件驱动总体设计的框图如图 2.2。

从框图的关系图中可以看出，各个管理模块之间是紧密联系的。内存管理是指处理器对内存资源的管理，内存资源包括数据，堆栈或程序代码。内存管理分为分段和分页两个部分，是最基本的模块，任何想要访问内存资源的操作都会使用内存管理。在多任务管理机制中，我们要保存一个任务的状态信息，调度任务执行，和切换不同的任务，这些都需要访问和读

写内存空间。尤其对于任务切换时，处理器要保存旧的任务状态段 TSS，然后加载新的任务状态段 TSS，这要遵循从段选择子到段描述符再到 TSS 的分段过程。

而对于中断和异常处理机制，我们需要将要处理器的中断服务程序和中断向量号对应，然后去处理多个中断或异常。当发生中断时，根据对应的中断向量号找到中断通道或陷阱通道，然后去执行对应的中断服务程序。这也是内存管理的分段过程。

多任务管理机制和中断或异常处理机制是相互包含的关系。当中断或异常发生时，有时需要处理器去执行中断服务程序，这实际上就是多任务之间的切换过程；同时在任务切换机制中隐含调用中断服务程序就是中断处理程序的响应过程。

对于多处理器管理机制，则是宏观上的多个处理器之间切换的过程，其原理其实和任务切换有类似的地方，比如也要加载和保存旧的处理器运行状态。

电源和功耗管理则是相对独立的部分，但是它会用到很多的系统资源作支撑。

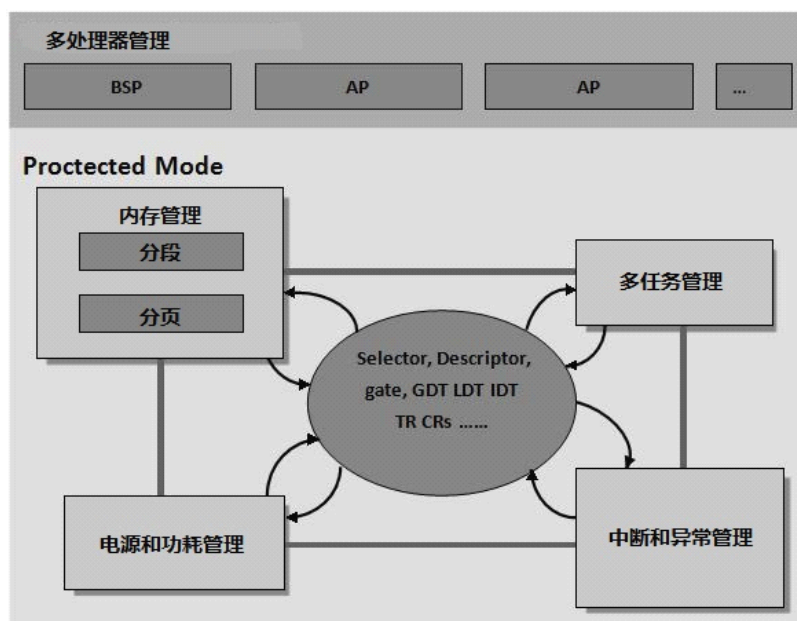


图 2.2 固件驱动总体设计框图

2.2 固件驱动各模块概述

2.2.1 内存管理

内存管理是对系统内存资源进行管理的机制，IA32 and 64 体系架构的内存管理分成两个部分：分段和分页^{[10][12]}。分段提供了将单独的代码，数据和堆栈隔离开的机制，这样多个任务可以互不干扰的运行在同一个处理器上。分页提供了实现传统分页需求的虚拟内存管理的机制，系统中程序执行环境的段按需的映射到物理内存中。

这两种机制都可以通过配置来支持单任务系统，多任务系统，或使用共享内存的多处理器系统。

分段机制将处理器可寻址的内存空间（线性地址空间）分成更小的受保护的地址空间，段。段中的任何一个字节的访问都要通过全局描述符表（GDT）或者局部描述符表（LDT）中的段描述符（segment descriptor）提供入口，即段的基地址。每个段描述符都有一个相关联的段选择子（segment selector），段选择子提供给软件相应的段描述符的偏移量，一个全局（或局部）标志位（决定选择字指向 GDT 或 LDT），和访问权限信息。

所以要访问一个段中的某个字节，必须将段选择子和偏移量提供给软件：段选择子提供了到达此段的段描述符的入口，处理器从段描述符中得到在线性地址空间中的段的基地址；偏移量提供了相对于段基址的字节位置。

分页支持一个“虚拟内存”的环境，它可以将一个很大的线性地址空间用小量的物理内存和一些磁盘模拟出来。使用分页时，每个段被分成页，这些页可能存储在物理内存中或磁盘上，OS 或管理程序保持着页路径和一组页表（page tables）来定位页的位置。当一个程序或任务尝试访问线性地址空间中的一个地址位置时，处理器使用页路径和页表来将线性地址转换成一个物理地址，然后执行一个请求的操作。

2.2.2 中断和异常管理

中断和异常管理是指系统处理突发事件的机制。中断和异常本质是事件，这些事件表明有需要处理器注意的情况存在于系统，处理器，或者当前正在执行任务的某个位置。它们一般会导致执行过程从当前运行的程序强制转移到中断处理程序或异常处理程序。中断和异常发生在程序执行过程的随机时间内，以响应硬件或软件发来的信号^[10]。

为了管理和响应异常和中断，每个在结构中定义的异常和中断情况都被分配了一个唯一的标识号，叫做向量号，允许的向量号的范围为 0~255。处理器会使用分配给异常或中断的向量号作为索引，索引到中断描述符表（IDT）中，该表给出了异常或中断处理程序的入口点。

当收到中断或检测到异常时，当前运行的程序或任务会被挂起同时处理器会转去执行中断或异常处理程序。当处理程序执行完之后，处理器将恢复被中断的程序或任务，被中断程序或任务的恢复要求保持程序的连续性。

2.2.3 多任务管理

一个任务就是一个工作单元，在这个工作单元里处理器可能被调度去执行指令或者被挂起。一个任务可以用来执行一段程序，一个进程，一个操作系统服务，或者一个中断或异常处理程序等等。IA32 and 64 架构提供了如下机制：保存一个任务的状态信息，调度任务执行，和切换不同的任务^[10]。

一个任务由两个部分组成：任务执行空间（task execution space）和任务状态段（task-state segment）。任务执行空间包括代码段，堆栈段和一个或多个数据段，如果操作系统使用了优先级保护机制，那么任务执行空间还要为每个优先级提供一个单独的堆栈。TSS 任务状态段定义了一个任务执行环境的状态，包括：一般用途寄存器，段寄存器，标志寄存器（EFLAGS register），EIP 寄存器，和带有堆栈指针的段选择子的状态，TSS 还包括此任务的 LDT 段选择子和分页结构层级的基地址^[11]。对于当前任务，TSS 的段选择子被保存在任务寄存器 TR 中，当切换一个任务时，两个任务的 TSS 等信息会发生切换：

（1）将当前任务的状态保存在当前 TSS 中；

（2）加载带有新任务段选择子的任务寄存器；

（3）通过 GDT 中的段描述符获得新的 TSS；

（4）从新的 TSS 中加载新任务的状态到一般用途寄存器，段寄存器，LDTR，控制寄存器 CR3（分页结构层级的基址），EFLAGS 寄存器，和 EIP 寄存器；

（5）开始执行新任务。

2.2.4 多处理器管理

当系统中有多个逻辑处理器时，就需要对多处理器进行管理。Intel IA32 and 64 架构提供了管理和提升连接在同一系统总线的多处理器的性能的机制，这就是多处理器机制，它主要包括以下内容^[10]：

（1）保持系统内存的联接——当两个以上的处理器同时尝试访问同一系统内存地址时，一些处理器之间的通信机制或内存访问协议将会用来保持数据的联接性。

（2）保持缓存的一致性——当一个处理器访问在另一个处理器上缓存的数据时，前者一定不能随意修改数据。如果它修改了数据，那么所有其他处理器都将访问修改过的数据。

（3）允许预先排好写入内存的顺序（memory ordering）——在很多情况下，内存写操作都要精确地遵循编程好的顺序。

(4) 在一组处理器之间处理多个中断应用请求——当多个处理器并行执行任务时，需要有一个集中的机制来接收中断并分配中断给多个处理器。

(5) 可以通过开发多线程和多进程来提升系统性能。

在多处理器机制中，一台电脑不再由单个 CPU 组成，而同时由多个处理器并行处理系统中的任务，共享内存和其他资源。系统将任务队列对称地分布于多个 CPU 之上，从而极大地提高了整个系统的数据处理能力。

2.3 开发环境及开发流程

2.3.1 EDKII 开发框架

EDKII (EFI Development Kit) 是一个开源的 UEFI BIOS 的发布框架，包括了一些由 Tiano 开发的 EFI 项目实例和大量基本的底层库函数，Intel 已经在 BSD 许可证下公开了其框架的所有代码。除此之外，EDK 同时也是一个开发、调试和测试 EFI 驱动程序、Option Rom 驱动和 EFI Shell 应用程序的综合平台，例如 Intel 在 EDK 框架包含了一些驱动程序二进制文件和开发示例，供 BIOS 生产厂商和研发机构继续开发^{[8][9]}。

本文中所研究的 Intel Xeon-E5 多处理器固件驱动开发是作为 EDKII 框架开发的一部分进行的，由于涉及知识产权的保护，在 EDKII 的发布版本中这部分代码并不是开源的，只是公布了其二进制文件。在 EDKII 开发过程中，处理器相关的开发代码主要包括在 IA32FamilyCpuPkg/ 目录中，该目录下主要包含了符合 UEFI 标准的处理器相关库文件以及 DXE 阶段的驱动程序代码，如图 2.3 所示。

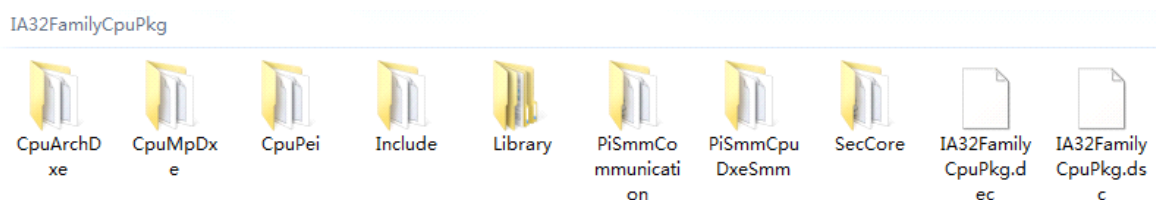


图 2.3 处理器固件驱动所包含的文件

2.3.2 开发调试过程

在使用 EDKII 开发之前，除了首先要更新源代码，同时还要进行开发环境的配置。首先介绍本项目中使用到的开发工具^{[13][14]}。

(1) 工程代码版本管理软件：TortoiseSVN 客户端

(2) 用来阅读和编辑源代码：Source Insight Program Editor

(3) 用于将源代码编译成二进制可执行文件: Microsoft Visual Studio 2008

(4) 用于查看系统调试信息的串口软件: Ttermpro

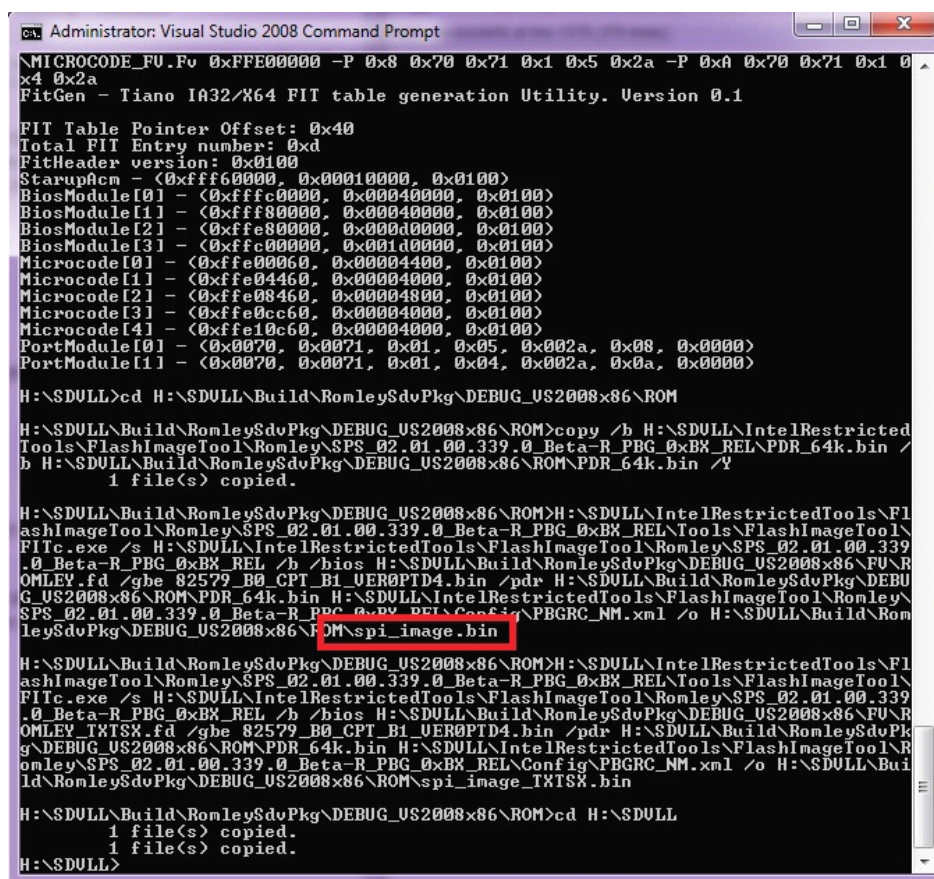
(5) 调试硬件和软件的 ITP 工具: Intel ITP Debugger^[13]

Intel ITP Debugger 是由 Intel 自主研发的一套用于调试 Intel X86 系统硬件设备和软件程序的工具, 它由 ITP 调试仪和 ITP 调试软件组成, 调试对象主要是 Intel X86 架构开发板。它借助专门的 ITP 调试仪通过 CPU 的专用信号引脚与 CPU 通信, 实施调试任务。

EDKII 项目中固件驱动的开发调试过程如下:

(1) 通过 TortoiseSVN 客户端更新 EDKII 开发框架, 并把源代码放在 C:\EDKII 目录下。

(2) 将当前源代码编译生成二进制可执行文件 spi_image.bin, 图 2.4 为 VS2008 编译的过程, 然后使用 flash 烧录器将 spi_image.bin 烧入开发板 flash 芯片中, 启动开发板, 以检查当前的源代码是没有任何 bug 的。



```

Administrator: Visual Studio 2008 Command Prompt

\MICROCODE_FU.Fw 0xFFE00000 -P 0x8 0x70 0x71 0x1 0x5 0x2a -P 0xA 0x70 0x71 0x1 0
x4 0x2a
FitGen - Tiano IA32/X64 FIT table generation Utility. Version 0.1

FIT Table Pointer Offset: 0x40
Total FIT Entry number: 0xd
FitHeader version: 0x0100
StartupAcm - (0xffff60000, 0x00010000, 0x0100)
BiosModule[0] - (0xfffc0000, 0x00040000, 0x0100)
BiosModule[1] - (0xffff80000, 0x00040000, 0x0100)
BiosModule[2] - (0xffe80000, 0x00040000, 0x0100)
BiosModule[3] - (0xffc00000, 0x001d0000, 0x0100)
Microcode[0] - (0xffe00060, 0x00004400, 0x0100)
Microcode[1] - (0xffe04460, 0x00004000, 0x0100)
Microcode[2] - (0xffe08460, 0x00004800, 0x0100)
Microcode[3] - (0xffe0cc60, 0x00004000, 0x0100)
Microcode[4] - (0xffe10c60, 0x00004000, 0x0100)
PortModule[0] - (0x0070, 0x0071, 0x01, 0x05, 0x002a, 0x08, 0x0000)
PortModule[1] - (0x0070, 0x0071, 0x01, 0x04, 0x002a, 0x0a, 0x0000)

H:\SDULL>cd H:\SDULL\Build\RomleySdvPkg\DEBUG_US2008x86\ROM

H:\SDULL\Build\RomleySdvPkg\DEBUG_US2008x86\ROM>copy /b H:\SDULL\IntelRestricted
Tools\FlashImageTool\Romley\SPS_02.01.00.339.0_Beta-R_PBG_0xBX_REL\PDR_64k.bin /
b H:\SDULL\Build\RomleySdvPkg\DEBUG_US2008x86\ROM\PDR_64k.bin /Y
1 file(s) copied.

H:\SDULL\Build\RomleySdvPkg\DEBUG_US2008x86\ROM>H:\SDULL\IntelRestrictedTools\FI
ashImageTool\Romley\SPS_02.01.00.339.0_Beta-R_PBG_0xBX_REL\Tools\FlashImageTool\
FITc.exe /s H:\SDULL\IntelRestrictedTools\FlashImageTool\Romley\SPS_02.01.00.339
.0_Beta-R_PBG_0xBX_REL /b /bios H:\SDULL\Build\RomleySdvPkg\DEBUG_US2008x86\FU\R
OMLEY.fd /gbe 82579_B0_CPT_B1_VER0PTD4.bin /pdr H:\SDULL\Build\RomleySdvPkg\DEBU
G_US2008x86\ROM\PDR_64k.bin H:\SDULL\IntelRestrictedTools\FlashImageTool\Romley\
SPS_02.01.00.339.0_Beta-R_PBG_0xBX_REL\Config\PBGRM.xml /o H:\SDULL\Build\Rom
leySdvPkg\DEBUG_US2008x86\ROM\spi_image.bin

H:\SDULL\Build\RomleySdvPkg\DEBUG_US2008x86\ROM>H:\SDULL\IntelRestrictedTools\FI
ashImageTool\Romley\SPS_02.01.00.339.0_Beta-R_PBG_0xBX_REL\Tools\FlashImageTool\
FITc.exe /s H:\SDULL\IntelRestrictedTools\FlashImageTool\Romley\SPS_02.01.00.339
.0_Beta-R_PBG_0xBX_REL /b /bios H:\SDULL\Build\RomleySdvPkg\DEBUG_US2008x86\FU\R
OMLEY.TXTSK.fd /gbe 82579_B0_CPT_B1_VER0PTD4.bin /pdr H:\SDULL\Build\RomleySdvPk
g\DEBUG_US2008x86\ROM\PDR_64k.bin H:\SDULL\IntelRestrictedTools\FlashImageTool\R
omley\SPS_02.01.00.339.0_Beta-R_PBG_0xBX_REL\Config\PBGRM.xml /o H:\SDULL\Bui
ld\RomleySdvPkg\DEBUG_US2008x86\ROM\spi_image.TXTSK.bin

H:\SDULL\Build\RomleySdvPkg\DEBUG_US2008x86\ROM>cd H:\SDULL
1 file(s) copied.
1 file(s) copied.
H:\SDULL>
  
```

图 2.4 编译生成可执行文件 spi_image.bin 的过程

(3) 用 Source Insight 软件创建一个 EDKII 框架的工程, 并把 EDKII 框架的源代码添加到 EDKII 的工程中。根据 IA32 and 64 相关的开发说明书和规范说明书, 修改和添加 IA32FamilyCpuPkg/目录下的处理器固件驱动相关的代码。

(4) 保存修改后的代码, 并按照 (2) 中的步骤编译代码, 生成目标可执行文件 spi_image.bin, 然后使用 flash 烧录器将 spi_image.bin 烧入开发板 flash 中。

(5) 将开发板通过 ITP 调试仪连接到调试机上, 本项目中开发板采用服务器开发板, 调试机为自己的已安装 ITP 调试软件的笔记本电脑。然后用 USB 转串口连接开发板和调试机。

(6) 给开发板上电, 打开 ITP 调试软件和串口 Ttermpro 软件, 配置相关参数, 如串口波特率和 ITP 中目标开发板的匹配。通过串口随时查看系统运行时打印的信息, 并使用 ITP 调试目标代码。由于编译生成的可执行二进制文件的文件格式是符合 Windows PE/COFF 标准的, 通过 ITP 调试软件可以定位本地调试机中的代码, 调试起来非常方便。如果看到串口软件中出错的信息, 就可以通过 ITP 软件中的 loadthis 命令打开此处的代码然后修改, 并根据修改后的代码直接继续运行, 直至运行通过。图 2.5 显示了根据串口调试信息查看 bug 原因, 并通过 ITP 软件调试的过程。

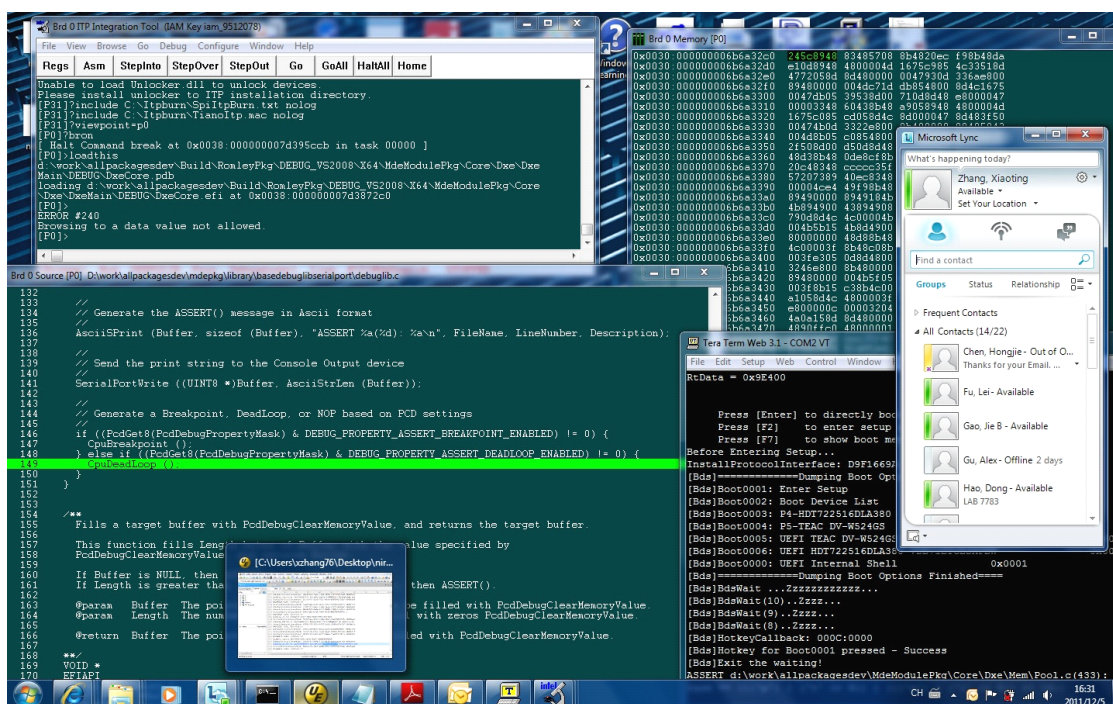


图 2.5 ITP 和 Ttermpro 软件调试过程

(7) 将调试通过的代码重新编译生成 spi_image.bin, 然后烧入开发板 flash 芯片, 查看开发板运行情况是否满足要求。如果满足就可以通过 SVN 客户端更新到服务器中, 不满足就在此基础上继续重复调试和修改代码, 直到成功。

2.4 项目实现目标

本论文设计要实现的就是 Intel Xeon-E5 多处理器的固件驱动开发, 主要实现其四个模块的开发: 内存管理, 中断和异常管理, 多任务管理, 以及多处理器管理。

对于内存管理部分, 将研究分段机制和分页机制实现原理和算法, 并通过伪代码形式阐述其具体实现; 中断和异常管理将会阐述系统管理中断和异常情况的机制, 通过伪代码具体

实现对一个中断服务程序管理的过程；多任务管理将对一个任务的执行过程给出原理性分析，并通过伪代码实现对多任务的管理过程；多处理器机制将重点分析多处理器初始化协议算法以及多处理器之间的通信机制，并结合伪代码给出其具体实现步骤。

2.5 本章小结

本章首先对本论文中研究的多处理器固件驱动开发项目作总体设计，阐述了系统需要管理的体系架构，并设计了固件驱动实现的总体框图。然后对固件驱动的各个模块做简要的介绍，给出各模块需要完成的工作。接着对项目的开发工具、环境及开发流程作了详细的介绍，具体阐述了 UEFI 固件驱动开发过程需要用到的开发工具以开发软件。在本章最后给出了 UEFI 固件驱动项目的实现目标。

第三章 内存管理

对于 IA32 and 64 架构的多处理器来说内存管理是基本的功能，处理器固件驱动中需要完成对内存管理机制的配置以及功能的实现。本章将详细描述内存管理功能中的分段机制和分页机制，并针对分段和分页的机制算法给出伪代码上的实现。

3.1 内存管理概述

IA32 and 64 架构多处理器的内存管理包括两个部分：分段（segmentation）和分页（paging）。分段提供了将单独的代码，数据和堆栈隔离开的机制，这样多个任务可以互不干扰的运行在同一个处理器上^{[10][12]}。分页提供了实现一个传统分页需求的虚拟内存系统的机制，系统中程序执行环境的段按需映射到物理内存中。分段和分页机制的原理如图 3.1 所示。

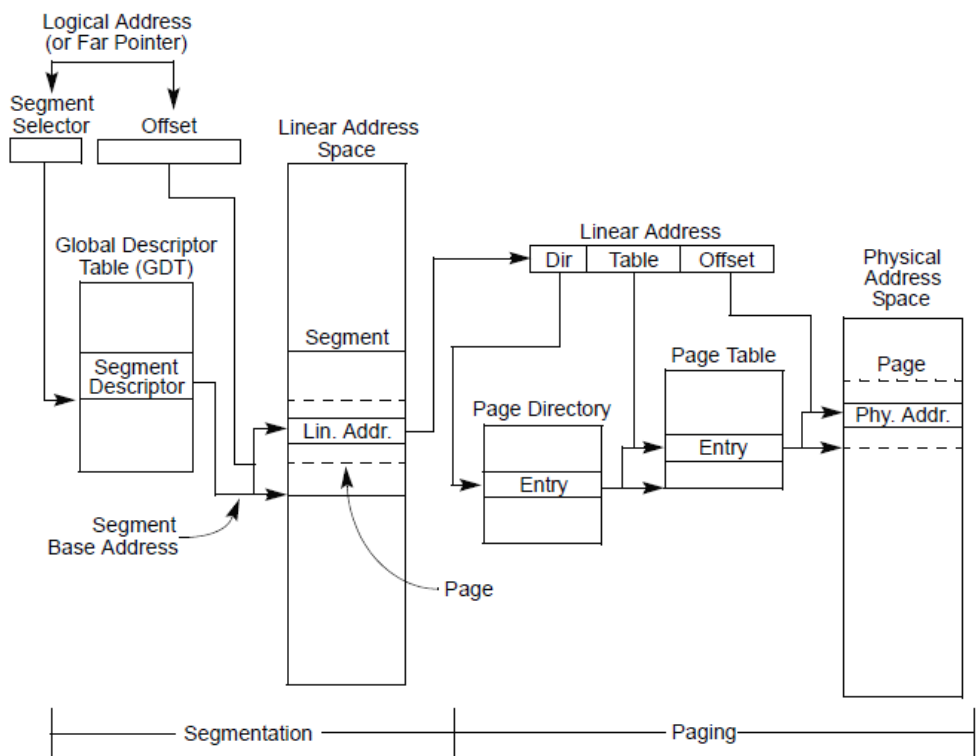


图 3.1 内存管理的分段和分页

如图所示，处理器使用分段机制将线性地址空间分成更小的受保护的地址空间，段（segments）。段用来保存一个程序的代码，数据，和堆栈，或者保存系统数据结构。如果多个任务运行在同一个处理器上，每个任务都能分配到自己的段，然后处理器会加强段之间的分界，保证程序之间互不干扰。分段机制还会允许各种段的分类，这样在特定类型段上执行

的操作会被限制在特定的段上^[10]。

系统中所有的段都包含在处理器的线性地址空间中，要定位一个特定段的一个字节，必须要提供逻辑地址。逻辑地址由段选择子和偏移量组成，段选择子是一个段的唯一识别符，其中包含了一个到描述符表（GDT）的偏移量，然后找到该段的段描述符。每个段都有一个段描述符，指定了段大小，访问权限和优先级，段类型，以及段的基地址。偏移量部分加上段基地址就可以定位一个段中的字节，所以基地址加偏移量就等于线性地址空间的线性地址。

物理地址空间定义为处理器在它的地址总线上能产生的地址范围，如果不使用分页，线性地址空间直接映射到处理器的物理地址空间。因为多任务系统通常能定义一个远大于它能包含的物理内存大小的线性地址，所以需要一种虚拟化线性地址空间的方法。这种线性地址虚拟化就是通过处理器的分页机制来实现的。

分页就是一个“虚拟内存”的环境，它能支持将一个很大的线性地址空间用小量的物理内存（RAM 和 ROM）和一些磁盘模拟出来^[15]。使用分页时，每个段被分成页，这些页可能存储在物理内存中或磁盘上。在执行程序中保持着页路径（page directory）和一组页表（page table）来定位页的位置，当一个程序或任务尝试访问线性地址空间中的一个地址位置时，处理器使用页路径和页表来将线性地址转换成一个物理地址，然后执行一个请求的操作。

如果当前要访问的页不在物理内存中，处理器会中断程序的执行，然后从磁盘中读取页到物理内存，并且继续执行程序。

3.2 系统分段机制模型

3.2.1 基本的内存平面模型

在内存管理的分段机制中，最简单的内存模型就是基本平面模型。在平面模型中执行程序可以访问连续的，未分段的地址空间。为了保证最大的扩展可能性，平面模型对系统设计者和程序员隐藏了分段机制。

为了实现基本平面模型，至少要创建两个段描述符：一个代码段，一个数据段，如图 3.2 所示^[12]。但是两个段都是映射到整个线性地址空间的，也就是说，两个段描述符有相同的基地址（0）和相同的 4GBytes 段大小限制。即使在要访问的地址没有物理内存，分段机制也不会产生超出限制的异常。ROM（EPROM）一般定位在物理地址空间的顶部，因为处理器在 FFFF_FFF0H 位置开始执行程序，RAM（DRAM）定位在地址空间的底部，因为处理器重启

后初始的 DS 数据段的基地址为 0。

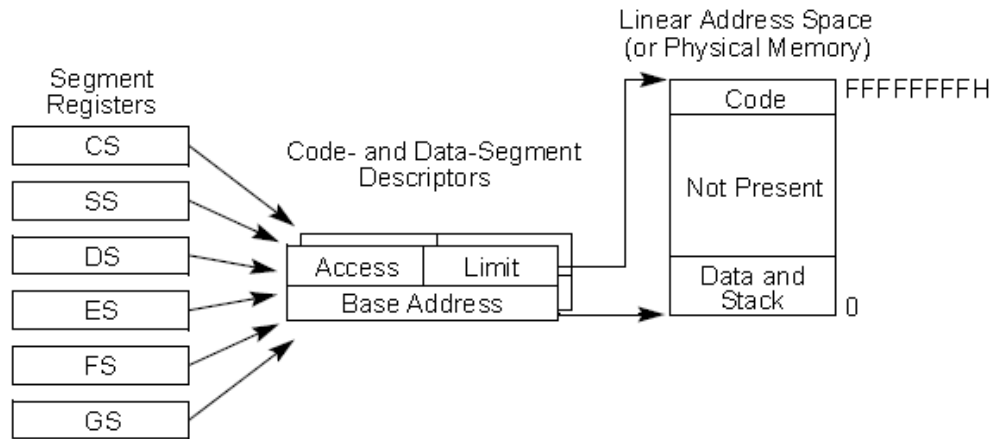


图 3.2 基本的平面模型

3.2.2 多重分段模型

平面模型是内存管理的基本模型，但是它并没有兼容分段机制，不能给线性地址空间和物理内存空间提供保护。多重分段模型具有分段机制的全面兼容，提供给硬件强制的代码，数据，程序和任务保护。这里，每个任务拥有自己的段描述符表和段，访问所有的段和单独的
程序执行环境是由硬件控制的。访问内存空间的字节时会进行访问检查，以防止引用超出段限制范围的地址，而且可以防止在特定的段上执行不允许的操作^[16]。多重分段模型如图 3.3 所示。

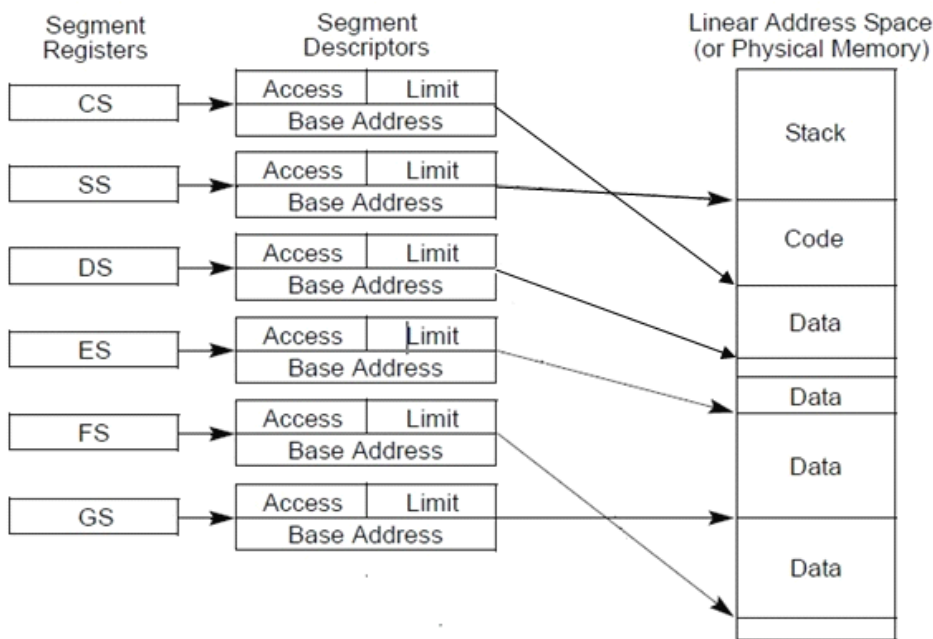


图 3.3 多重分段模型

3.3 分段机制

3.3.1 段选择子和段寄存器

段选择子是一个段的 16bit 标识符，它不直接指向段，而是指向定义此段的段描述符（图 3.1）。段选择子作为一个指针变量，对应用程序是可见的。段选择子包含 Index，TI 和 RPL 字段：

- （1）Index（bits 3-15）用来选中 GDT 或 LDT 的 8192 个描述符中的一个。
- （2）TI（table indicator）flag（bit 2）指定了使用的描述符表：GDT（0），LDT（1）。
- （3）Requested privilege level（RPL）（bit 0 和 1）指定了选择子的权限级（0-3），0 为最高权限级。

当执行指定的程序时，段选择子的内容要被加载至段寄存器中，处理器提供了容纳 6 个段选择子的段寄存器组：CS、SS、DS、ES、FS、GS，图 3.4 为 CS 寄存器。每个段寄存器支持一个特定的内存参量类型（代码，堆栈，或数据），对于实际执行程序，至少要加载代码段，数据段，和堆栈段寄存器^{[10][12][16]}。

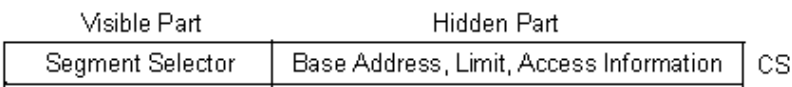


图 3.4 段寄存器组

对于一个正在执行的任务，它可能会尝试访问代码段，数据段或堆栈段，这时对应段的段选择子必须要被加载进段寄存器中。所以，即使系统定义了上千个段，但是对于即时只能获取 6 个。当段选择子被加载进段寄存器的可视部分时，处理器也会将段选择子指向的段描述符中的基地址，段大小，和访问信息加载进段寄存器的隐藏部分，当前任务会根据图 3.3 所示的方向找到对应的段。

3.3.2 段描述符和段描述符表

段描述符是存在于段描述符表的一个数据结构，提供了一个段的位置和大小，还有访问控制和状态信息。通过段描述符可以索引到指定的段，图 3.5 是一个段描述符的格式。

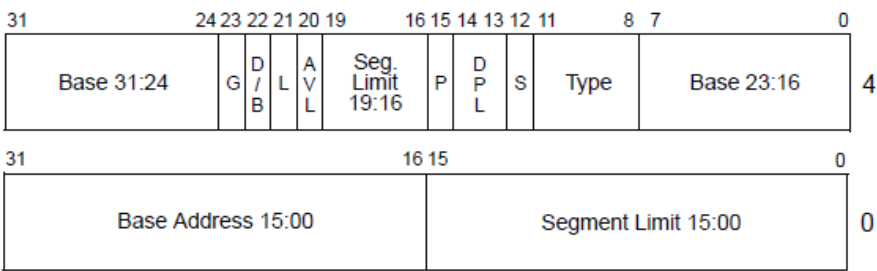


图 3.5 段描述符格式

段描述符中有一些关键的字段用来描述段的属性。

- (1) Segment limit 字段指定了段的大小限制。
- (2) Base 字段定义了该段的字节 0H 所在的位置，即段的基地址。
- (3) S 标志位指定该段是否是系统段。
- (4) DPL 字段指定了段的权限级，用来控制对段的访问。
- (5) Type 字段指定了段的类型。当 S 标志位不同时，即对于段描述符和系统段描述符，type 表示的含义不同，该字段通常与 P，DPL，S 同时使用，如后面的伪代码实现。
- (6) P 标志位表示该段是否已存在于内存中。

当段描述符中的 S 标志位被置 1 时，该段为代码段或数据段。当段描述符中的描述符类型标志位 S=0 时，该段描述符为一个系统描述符。处理器定义了如下系统描述符：

- (1) LDT (local descriptor table) 段描述符
- (2) TSS (task-state segment) 描述符
- (3) 调用通道 (call-gate) 描述符
- (4) 中断通道 (interrupt-gate) 描述符
- (5) 陷阱通道 (trap-gate) 描述符
- (6) 任务通道 (task-gate) 描述符

这些描述符类型分成两类：系统段描述符和通道描述符。系统段描述符指向了系统段 (LDT 和 TSS 段)，通道描述符通向它们自己的“通道”，通道是指程序或异常处理程序 (函数) 的入口点，如图 3.6。

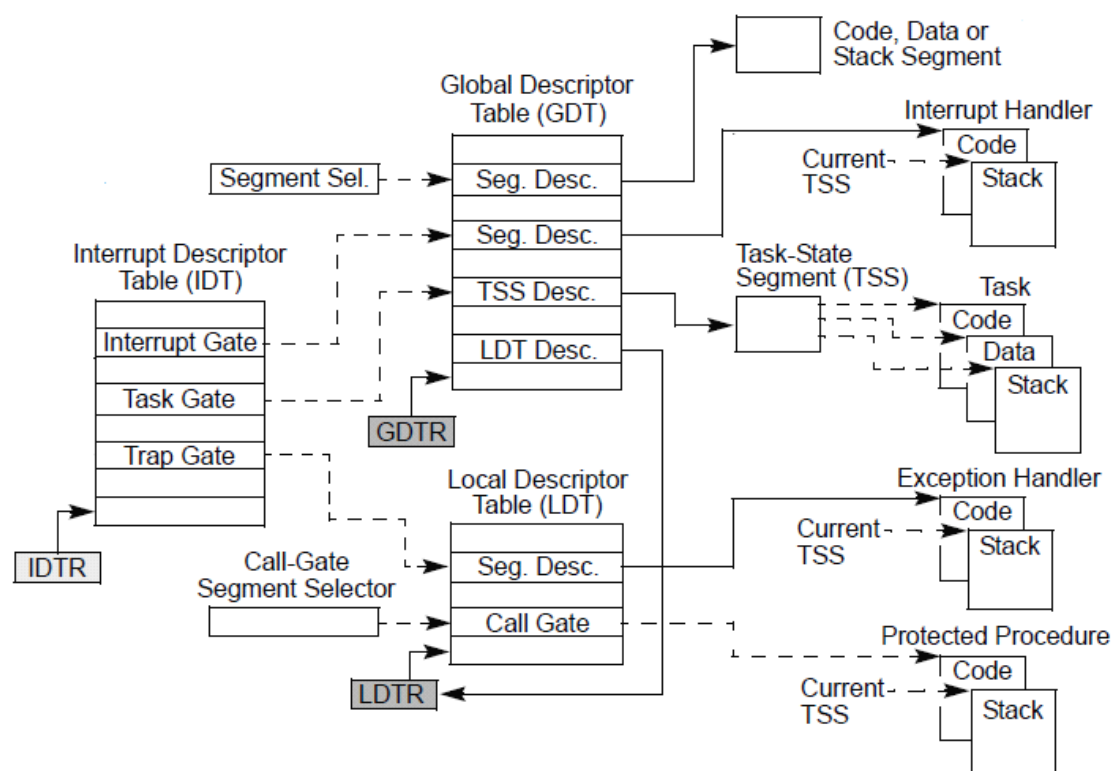


图 3.6 段描述符和段描述符表

段描述符表就是一组段描述符，它分为全局描述符表 GDT 和局部描述符表 LDT。一个段描述符表是长度可变的，并且包含最多 8192 个 8-byte 描述符。每个系统都会定义一个 GDT，所有的程序和任务都可使用它。但是视情况可以选择定义一个或多个 LDT，例如，可以为一个正在运行的任务定义一个或几个 LDT，或者所有的任务共享相同的 LTD。图 3.6 中描述了段描述符表以及段描述符之间的引用和包含关系。

3.3.3 逻辑地址到线性地址的转换

在保护模式的系统架构级别，处理器通过两个阶段的地址转换到达物理地址：逻辑地址转换和线性地址空间分页。即使使用最少的段，处理器地址空间的每个字节也是通过一个逻辑地址访问的。逻辑地址由一个 16bits 的段选择子和一个 32bits 的偏移量组成，如图 3.7 所示，段选择子确定了此字节所在的段，偏移量确定了相对于段基地址的字节位置。

如果不使用分页，处理器会将线性地址直接映射到物理地址，即线性地址直接从地址总线上出来。如果使用分页，则需要第二级转换将线性地址转换到物理地址。

处理器会将每个逻辑地址都转换成线性地址，线性地址是存在于线性地址空间的一个 32bit 地址。就像物理地址空间一样，线性地址空间也是平面的，共有 2^{32} 字节(0~FFFFFFFFH)。线性地址空间中包含了系统定义的所有的段和系统表。

处理器会按照如下步骤将一个逻辑地址转换到一个线性地址^{[10][16]}：

- (1) 将段选择子加载至段寄存器中，通过段选择子的 index 定位 GDT 或 LDT 中的段描述符，并将其读入处理器中。
- (2) 处理器检查段描述符的访问权限和段大小限制以确保段是可以访问的。
- (3) 将段描述符中的基地址加上逻辑地址的偏移量就会得到线性地址。(如图 3.7)

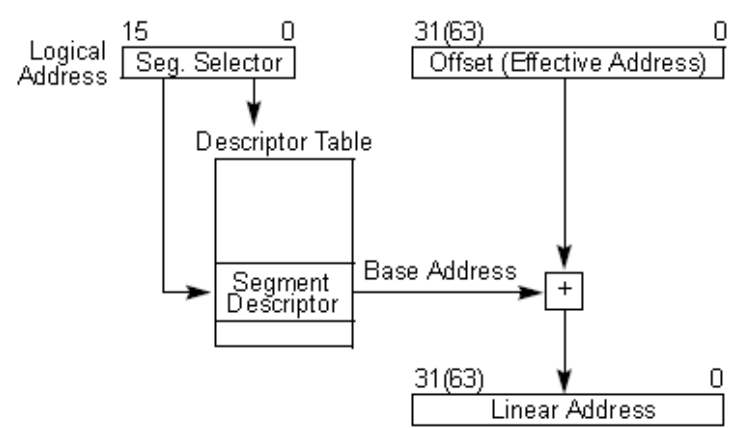


图 3.7 逻辑地址到线性地址的转换

3.3.4 分段机制的算法实现

从前面几节的分段机制描述中可以看出，如果想要访问线性地址空间中的地址，必须将段选择子和偏移量作为参数传给程序入口。但是在固件驱动的设计中，我们需要为每个程序初始化其所需的段描述符表。当程序被执行时，它可以通过表中的段描述符入口找到对应的段，然后找到段中指定的线性地址。

接下来给出的伪代码实现中，我们假设当前只有一个程序等待执行，且以 GDT 为例。在本实现例程中，我在 GDT 中分别定义了线性数据段，程序代码段，程序数据段，堆栈段，以及 LDT 系统段的描述符，有时还会定义一个额外的数据段留作备用。

- (1) 首先根据图 3.5 的段描述符格式，将 GDT 中的段描述符定义如下。

```
typedef struct _GDT_DESCRIPTOR {
    UINT16    Limit15_0;
    UINT16    Base15_0;
    UINT8     Base23_16;
    UINT8     Type;
    UINT8     Limit19_16_and_flags;
    UINT8     Base31_24;
} GDT_ENTRY;
```

(2) 然后定义该程序中需要的段的段描述符。

```
typedef struct _GDT_DESCRIPTOR {
    GDT_ENTRY      Null;    //第一个描述符必须为空
    GDT_ENTRY      Linear;
    GDT_ENTRY      ProCode;
    GDT_ENTRY      ProData;
    GDT_ENTRY      ProStack;
    GDT_ENTRY      SysDesc;
    GDT_ENTRY      SpareDesc;
} GDT_ENTRIES;
```

(3) 为这些段描述符定义段选择子。

```
#define OFFSET_OF(TYPE, Field) ((UINTN) &(((TYPE *)0)->Field))
//段描述符在 GDT 中的偏移位置即段选择子的值

#define NULL_SEL          OFFSET_OF (GDT_ENTRIES, Null)
#define LINEAR_SEL        OFFSET_OF (GDT_ENTRIES, Linear)
#define PRO_CODE_SEL      OFFSET_OF (GDT_ENTRIES, ProCode)
#define PRO_DATA_SEL      OFFSET_OF (GDT_ENTRIES, ProData)
#define PRO_STACK_SEL     OFFSET_OF (GDT_ENTRIES, ProStack)
#define SYS_DESC_SEL      OFFSET_OF (GDT_ENTRIES, SysDesc)
#define SPARE_DESC_SEL    OFFSET_OF (GDT_ENTRIES, SpareDesc)
```

(4) 创建程序的 GDT。

```
STATIC GDT_ENTRIES GdtTable = {
    { 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, }, // NULL_SEL, NULL 段必须为全 0
    // LINEAR_SEL
    { 0x0FFFF, // limit 0xFFFFF
      0x0,      // base 0
      0x0,
      0x092,    /* P(present)=1; DPL=0; S=1(非系统段描述符); Type=0010B,表示
                  为数据段, 且可读写*/
      0x0CF,    // page-granular, 32-bit
      0x0, },
```

```

// PRO_CODE_SEL
{ 0x0FFFF, 0x0, 0x0,
  0x09B,           //Type=1011B, 表示为代码段, 可读可执行可访问
  0x0CF, 0x0, },
// PRO_DATA_SEL
{ 0x0FFFF, 0x0, 0x0, 0x092, 0x0CF, 0x0, },
// PRO_STACK_SEL
{0x0FFFF, 0x0, 0x0, 0x09A, 0x0CF, 0x0, },
// SYS_DESC_SEL
{ 0x0FFFF, 0x0, 0x0,
  0x082,
  // P(present)=1; DPL=0; S=0(系统段描述符); Type=0010B,表示为 LDT 段描述符
  0x0CF, 0x0, },
  // SPARE_DESC_SEL, 额外段描述符,留作备用
{ 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, },
};

```

(5) 初始化已经定义的 GDT 并生成段选择子, 使程序可以通过段选择子使用该 GDT。

```

VOID InitGlobalDescriptorTable (VOID )
{
  //定义 GDT 描述符表和 GDT 寄存器
  GDT_ENTRIES *gdt;
  IA32_DESCRIPTOR gdtPtr;
  //为 gdt 分配运行时的数据池
  gdt =分配运行时数据区;
  将 gdt 以 8bytes 方式对齐;
  //将已定义的 GDT 拷贝至 gdt 中
  CopyMem (gdt, &GdtTable, sizeof (GdtTable));
  //设置 GDTR 的基地址和段大小限制, 并写入 GDTR 中
  gdtPtr.Base = (UINT32)(UINTN)(VOID*) gdt;
  gdtPtr.Limit = (UINT16) (sizeof (GdtTemplate) - 1);
  AsmWriteGdtr (&gdtPtr);
}

```

根据段描述符偏移量得到段选择子，并将其写入段寄存器中;

}

3.4 分页机制

前两节中描述了系统使用的分段模型，分段机制的原理，以及如何使用分段机制将逻辑地址转换成线性地址的过程。分页则是转换线性地址的过程，转换后的物理地址可以用来访问内存空间或 I/O 设备。分页将线性地址转换成物理地址，并且决定每次转换过程中线性地址的访问权限和内存类型^[10]。

3.4.1 分页模式以及模式切换

在 IA32 and 64 架构中，处理器使用 3 种分页模式：32-bit 分页，PAE 分页，IA-32e 分页（表 3.1）。这 3 种模式的控制和切换是通过如下比特位实现的：

- (1) 控制寄存器 CR0 中的 WP 和 PG 标志位 (bit16, bit31);
- (2) 控制寄存器 CR4 的 PSE, PAE, PGE 和 PCIDE 标志位 (bit4, bit5, bit7, bit17);
- (3) IA32e 模式限定寄存器 IA32_EFER 的 LME 和 NXE 标志位 (bit8 和 bit11)。

表 3.1 不同分页模式的特性

Paging mode	CR0.PG	CR4.PAE	LME in IA32_EFER	Linear-address width	Physical-address width	Page size(s)	Supports execute-disable?
None	0	N/A	N/A	32	32	N/A	No
32-bit paging	1	0	0	32	Up to 40	4-KByte 4-MByte	No
PAE paging	1	1	0	32	Up to 52	4-KByte 2-MByte	Yes
IA-32e paging	1	1	1	48	Up to 52	4-KByte 2-MByte 1-GByte	Yes

如表 3.1 所示，如果 CR0.PG=0，表示不使用分页机制，逻辑处理器将所有的线性地址都当作物理地址。CR4.PAE 和 IA32_EFER.LME 都将被处理器忽略，同样还有 CR0.WP, CR4.PSE, CR4.PGE 和 IA32_EFER.NXE。如果 CR0.PG=1，分页有效，同时必须保证 CR0.PE=1 (protection enable)。如果分页有效，CR4.PAE 和 IA32_EFER.LME 的值决定使用的分页模式。所以通过设置相应的比特位就可以实现 3 中模式的切换。

这 3 种分页模式在下面的细节中不同（如表 3.1）：

- (1) 线性地址位宽，等待转换的线性地址的位宽大小。
- (2) 物理地址位宽，分页产生的物理地址的位宽大小。
- (3) 页面大小，转换的线性地址间隔。同一页的线性地址转换到相同页的物理地址。
- (4) 对禁止执行访问权限的支持，是指该页可以有访问的权限，但是禁止执行，这样做以保护数据。一些分页模式下，程序可能被禁止从可读页中取指令。

3.4.2 分页模式控制比特位

在 3 中分页模式中，以下控制比特位控制每个分页模式如何操作，它们对我们编写固件驱动代码很有帮助。

(1) CR0.WP, bit 16

CR0.WP 可以保护页面不被超级用户模式 (supervisor-mode) 写入。如果 WP=0，运行在超级用户模式的 (权限级 CPL<3) 的程序可以对只读访问权限的线性地址进行写操作；如果 WP=1，不能写操作。

(2) CR4.PSE, PGE, PCIDE, bit 4、7、17

CR4.PSE 可以使 32-bit 分页模式的 4-MByte 页有效。如果 CR4.PSE=0，32-bit 分页只可以使用 4-KByte 页；如果 PSE=1，32-bit 分页可以使用 4-KByte 页和 4-MByte 页。

CR4.PGE 控制全局页，全局页用于不同地址空间之间共享转换信息。如果 CR4.PGE=1，可以将指定的转换过程在地址空间之间共享。

CR4.PCIDE 控制 IA-32e 分页模式的转换过程内容标识符 (process-context identifiers, PCIDs)，即只在 IA-32e 分页模式下 PCIDE=1。PCIDs 可以使一个逻辑处理器缓存多个线性地址空间的信息。

(3) IA32_EFER MSR.NXE, bit 11

IA32_EFER.NXE 控制 PAE 分页和 IA-32e 分页的禁止执行访问权限。如果 NXE=1，指定线性地址中的指令将被保护。

对于 32-bit 分页，分页允许程序从任何可读的线性地址中取指令，NXE 对其无影响。如果要限制程序从可读页取指令，就必须使用 PAE 分页或 IA-32e 分页。

3.4.3 分页机制原理：分层级分页

在 3.4.1 节中描述了分页的 3 种模式以及控制比特位，在 IA32 and 64 架构处理器中所有的 3 种分页模式都使用分层级的分页机制，这种分页机制是通过分层级分页结构体

（Hierarchical Paging Structures）实现的^{[10][17]}。本节给出分层级分页机制的概述，接下来会给出 32-bit 分页模式的分页机制细节。

每一个分页结构体都为 4096 Bytes，由单独的入口（entry）组成。对于 32-bit 分页，每个入口是 32bits（4bytes），因此每个分页结构都有 1024 个入口；对于 PAE 分页和 IA-32e 分页模式，每个入口是 64bits（8bytes），因此每个分页结构共有 512 个入口。注意对于 PAE 分页有个例外：如果一个分页结构是 32bytes，那么它只包含 4 个 64-bits 入口。

如图 3.8 所示，处理器使用线性地址的高位部分确定一连串分页结构体入口，这些入口的最后一个入口确定此线性地址对应的物理地址所在的页面（page frame）。线性地址低位部分的页偏移量（page offset）确定页面的最后物理地址。

每个分页结构体入口包含一个物理地址，这个物理地址可能是另一个分页结构体的地址或是一个页面的地址。前一种情况叫做此入口引用（reference）到其他分页结构，后一种情况叫做此入口映射（map）到一个页。

任何分页模式中第一个分页结构的基地址都位于 CR3 中。一个线性地址的转换过程就是下面的迭代过程：最高几位选中使用 CR3 定位的分页结构体中的一个入口；如果入口引用到另一个分页结构，转换过程将用接下来的字段继续上面的操作；如果入口映射到一个页，那么入口中的物理地址就是页面的地址，剩下的低位就是页偏移量^[18]。

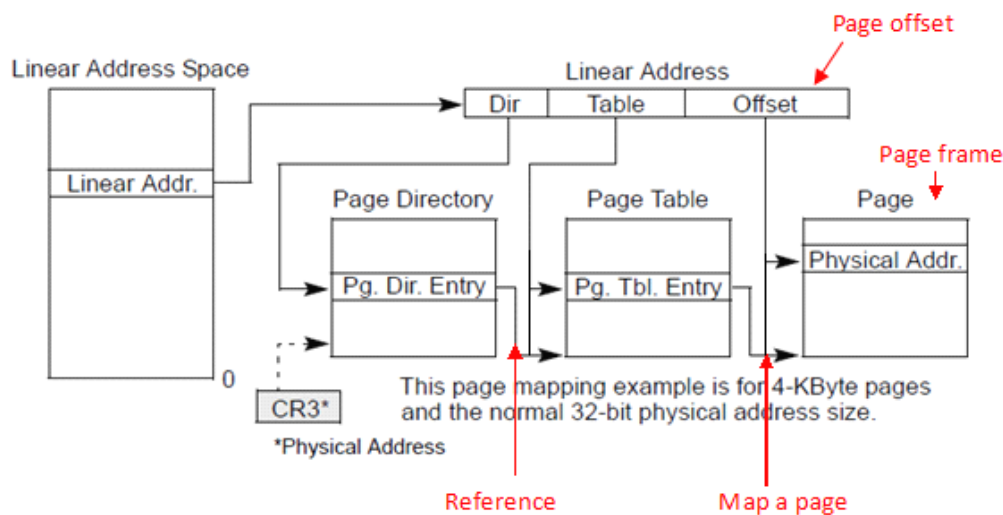


图 3.8 分层级分页机制

分页结构体基于它们在转换过程中的用处分别有不同的名字。表 4-2 给出了不同的分页结构体的名字，并给出了用来定位它们的物理地址来源（从 CR3 中或另一个分页结构体入口中）。表中还给出了对应于此分页结构体，线性地址中用来选中其中一个入口的比特字段，还有最后一个入口会映射一个多大的页面。

表 3.2 不同分页模式中的分页结构

分页结构体名	入口名	分页模式	物理地址来源	选中入口的字段	映射的页大小
PML4 table	PML4E	32-bit, PAE	N/A		
		IA-32e	CR3	47:39	N/A(PS must be 0)
page-directory-pointer table	PDPTE	32-bit	N/A		
		PAE	CR3	31:30	N/A(PS must be 0)
		IA-32e	PML4E	38:30	1-GByte page if PS=1
Page directory	PDE	32-bit	CR3	31:22	4-MByte page if PS=1
		PAE, IA-32e	PDPTE	29:21	2-MByte page if PS=1
Page table	PTE	32-bit	PDE	21:12	4-Kbyte page
		PAE, IA-32e		20:12	4-Kbyte page

注意:

PS (page size) 是当前入口中的 bit7, 它可以决定剩余的 bit 位是引用到下一个分页结构还是映射到页。32-bit 分页模式下, 如果 CR4.PSE=1, 当 PDE 的 PS(bit7)=1 时, 剩余的 21:0 全部用来映射到一个 4-MByte 页; 对于 PAE 分页模式, PDPTE 中的 PS 必须等于 0, 但是当 PDE 的 PS=1 时, 剩下的 20:0 用来映射到一个 2-MByte 页; 对于 IA-32e 分页模式, 如果 PDPTE 的 PS=1, 则剩下的 29:0 用来映射到一个 1-GByte 页, 否则如果 PDE 的 PS=1, 则剩下的 20:0 用来映射到一个 2-MByte 页。而且对 PAE 和 IA-32e 分页, 它们不受 CR4.PSE 的影响。

3.4.4 32-bit 分页以及算法实现

3.4.2 节中对分层级的分页机制做了概述, 本节将对 32-bit 分页进行详细的研究, 给出物理地址的转换过程, 接下来将对转换过程给出算法上的伪代码实现。对于其他分页模式的原理都与 32-bit 分页模式类似, 本论文中不做研究。

如 3.4.1 节中描述, 如果 CR0.PG=1, CR4.PAE=0, 逻辑处理器这时使用 32-bit 分页模式。32-bit 分页会将 32-bits 线性地址转换到 40-bits 物理地址, 即使 40-bits 对应 1TBytes, 但是线性地址还是被限制在 32bits, 即可访问的最大线性地址空间是 4GBytes。根据分页结构体入口 PDT 的 PS 标志位的设置, 32-bit 分页可以映射到一个 4-Kbyte 页面或 4-Mbyte 页面, 如表 3.2。图 3.9 和图 3.10 分别是映射 4-Kbyte 页面和 4-Mbyte 页面的原理图^[10]。

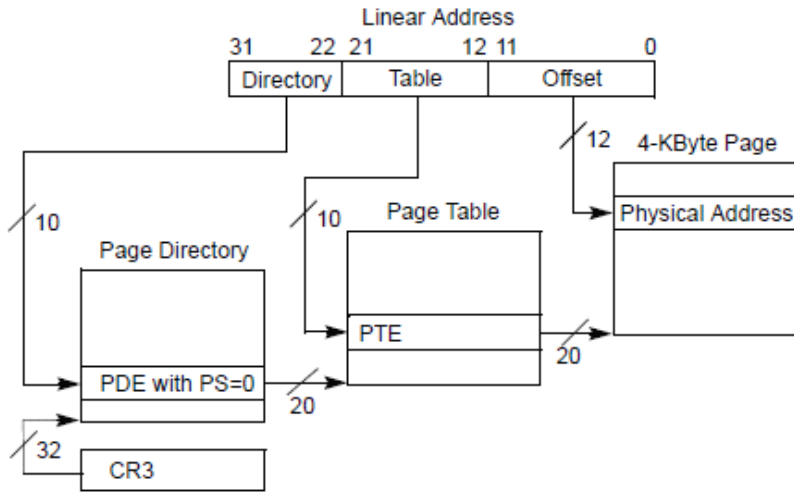


图 3.9 使用 32-bit 分页映射到 4-KByte 页

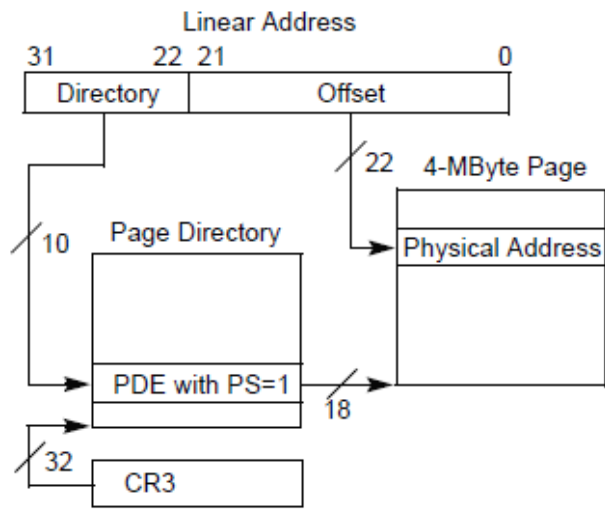


图 3.10 使用 32-bit 分页翻译到 4-MByte 页

将一个线性地址转换到物理地址要按照下面的步骤完成：

（1）首先使用 CR3 的 32:12 比特位定位第一个分页结构体——页路径表（page directory），页路径表是 4-KByte 对齐的，也就是一个页路径表大小是 4-KByte。一个页路径由 1024 个 32-bit 入口（PDE）组成，页路径中的一个 PDE 是由下面定义的物理地址确定的：

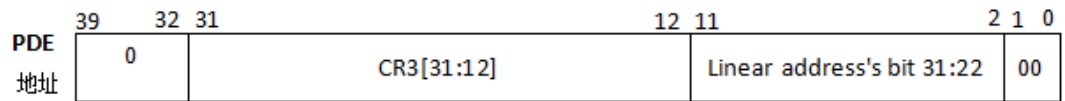


图 3.11 PDE 的地址

（2）因为一个 PDE 只由线性地址的 bits 31:22 来确定，剩余的 21:0 对应着 4-MByte(2^{22}) 的线性地址空间，所以它控制着对这 4-MByte 区域的访问。CR.PSE 和 PDE 的 PS 标志位(bit7) 决定了 PDE 的使用。

如果 CR4.PSE=1，PDE.PS=1，PDE 映射到一个 4-MByte 页，最后的物理地址：

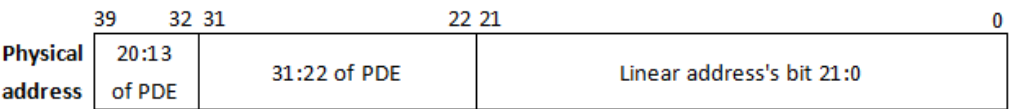


图 3.12 页面地址

如果 CR4.PSE=0，或 PDE.PS=0，PDE 的 bits 31:12 用来定位一个 4-KByte 自动对齐的页表（page table）。一个页表也是由 1024 个 32-bits 入口（PTEs）组成，一个 PTE 通过下面的物理地址确定：

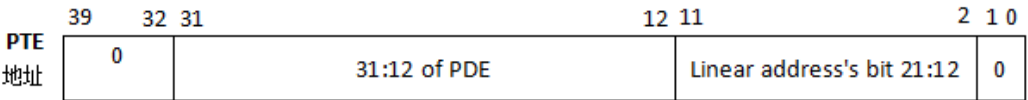


图 3.13 PTE 的地址

因为一个 PTE 只由线性地址的 bits 31:12 确定，线性地址还剩下 bits 11:0，所以一个 PTE 映射一个 4-KByte 页（图 3.9）。计算最后的物理地址：



图 3.14 页面地址

表 3.3 中给出了 CR3 和分页结构体入口的格式。分页结构入口包括 PDE 和 PTE，它们可能映射到一个页面，也可能引用到下一个分页结构体的入口，也可能两者都不是的。CR3、PDE 和 PTE 中的 bit0(p)和 bit7(PS)决定了一个入口如何使用，所以它们在图中被高亮。

表 3.3 CR3 和分页结构体入口

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
Address of page directory ¹																Ignored				P C D	P W T	Ignored		CR3								
Bits 31:22 of address of 2MB page frame						Reserved (must be 0)				Bits 39:32 of address ²		P A T	Ignored	G	1	D	A	P C D	P W T	U / S	R / W	1	PDE: 4MB page									
Address of page table												Ignored				0	I g n	A	P C D	P W T	U / S	R / W	1	PDE: page table								
Ignored																						0			PDE: not present							
Address of 4KB page frame												Ignored				G	P A T	D	A	P C D	P W T	U / S	R / W	1	PTE: 4KB page							
Ignored																						0			PTE: not present							

上述内容中给出了 32-bit 分页模式的原理，下面将根据上述步骤给出该机制的算法实现。

(1) 首先根据表 3.3 定义 3 个分页结构体入口。

//定义 4K 页面的 PDE

```
typedef struct {
    UINT32  Present:1;           // 0 = Not present in memory, 1 = Present in memory
    UINT32  ReadWrite:1;         // 0 = Read-Only, 1= Read/Write
    UINT32  UserSupervisor:1;    // 0 = Supervisor, 1=User
    UINT32  WriteThrough:1;      // 0 = Write-Back caching, 1=Write-Through caching
    UINT32  CacheDisabled:1;     // 0 = Cached, 1=Non-Cached
    UINT32  Accessed:1;          // 0 = Not accessed, 1 = Accessed (set by CPU)
    UINT32  MustBeZero:3;        // Must Be Zero, bit7(PS)=0
    UINT32  Available:3;         // Available for use by system software
    UINT32  PageTableBaseAddress:20; // Page Table Base Address
} IA32_PAGE_DIRECTORY_ENTRY_4K;
```

//同理定义 4M 页面的 PDE 和 4K 页面的 PTE

```
typedef struct {
    .....
    UINT32  PageSize:1;          // PS 标志位必须设置为 1
    .....} IA32_PAGE_DIRECTORY_ENTRY_4M;
typedef struct {.....} IA32_PAGE_TABLE_ENTRY_4K;
```

(2) 为指定的页面地址创建 4K 页面的页表。

```
VOID Create4KPageTables (  UINT32  *PageTable , UINT32  PageAddress)
```

//PageTable 是页表的基地址, PageAddress 为最后的页面地址

```
{
    IA32_PAGE_DIRECTORY_ENTRY_4K          *PDE_4KB;
    IA32_PAGE_TABLE_ENTRY_4K              *PTE_4KB;
```

(a) PTE_4KB = (IA32_PAGE_TABLE_ENTRY_4K *) (PageTable+0x1000);

//每个页表都是 4K 大小

```
PDE_4KB = (IA32_PAGE_DIRECTORY_ENTRY_4K *)PageTable; //PDE 指向页表
```

(b) //设置 PDE 的内容, 使之能索引到页表

```
*PDE_4KB = (UINT32) PTE_4KB; //
```

```
(*PDE_4KB).ReadWrite = 1;
```

```
(*PDE_4KB).Present = 1; //Present 表示已存在于内存中
```

```
(*PDE_4KB).MustBeZero = 1; // 1 = 001B, 即 PS(bit 7) = 0
```

(c) //根据页面地址填充这个页表的每个 PTE 中的内容，共有 1024 个 PTE

```
for (int i = 0; i < 1024; i++, PTE_4KB++) {
    *PTE_4KB = (UINT32)PageAddress;
    (*PTE_4KB).ReadWrite = 1;
    (*PTE_4KB).Present = 1;
    PageAddress += 0x1000; //因为页面是 0x1000=4K 大小, 所以每次要自加 0x1000
}
}
```

(3) 根据上面的步骤同样可以实现 4M 页面的页表的创建，需要注意的是 4M 页面 PDE 的某些比特位的设置。这里只给出填充每个 PDE 内容的伪代码语句。

```
VOID Create4MPageTables (  UINT32  *PageTable , UINT32  PageAddress)
{
    .....
    for (int i = 0; i < 1024; i++, PDE_4MB++) {
        (*PDE_4MB) = (UINT32)PageAddress;
        (*PDE_4MB).ReadWrite = 1;
        (*PDE_4MB).Present = 1;
        (*PDE_4MB).PageSize = 1; // PS 标志位必须设置为 1
        PageAddress += 0x400000; //因为页面是 0x400000=4MB 大小
    }
}
```

3.5 本章小结

本章的内容主要讨论了处理器的内存管理功能，对内存管理功能作了概述，并对内存管理的分段机制和分页机制作了研究和实现。

基于 IA32 and 64 架构处理器的内存管理功能包括两个重要的机制：分段机制和分页机制。分段机制用来实现从逻辑地址到线性地址的转换，通过逻辑地址的段选择子和偏移量定位线性地址空间的目标地址。在对分段机制的研究中，首先阐述了 Intel Xeon-E5 多处理器所采用的多重分段模型。然后对分段机制的原理作了详细研究，重点介绍了如何通过段选择子

和段描述符实现从逻辑地址到线性地址空间的线性地址的转换的过程。最后通过对分段机制算法的分析，并设计用伪代码实现了一个程序的 GDT 的创建，并定义了线性数据段，程序代码段，程序数据段，堆栈段，以及 LDT 系统段的描述符。

分页机制用来实现从线性地址空间的线性地址到物理地址空间的物理地址的转换。它使用分层级的分页机制，通过多个分层级分页结构体最终映射到物理地址空间的页面，并得到最终的物理地址。在对分页机制的研究中，首先阐述了分页的 3 种模式以及转化关系，并介绍了影响分页机制的某些比特位。然后详细分析了分页机制的原理，对 32-bit 分页模式的算法作了重点研究。最后通过伪代码实现了 32-bit 分页的给定页面地址和页表之间的映射，并为页面地址创建了所有的 PTE。

第四章 中断和异常管理

本章将描述运行在保护模式下的 IA32 and 64 架构处理器的中断和异常管理机制，重点将研究机制实现的原理，并基于对原理的算法分析用伪代码实现中断和异常管理功能。

4.1 中断和异常管理概述

中断和异常管理功能是处理器为了管理系统内外特殊情况的机制。它将结构化定义的中断或异常情况与中断向量号相对应，然后通过内存管理机制索引到中断或异常处理程序，最后通过执行中断或异常处理程序实现中断或异常情况的响应^{[10][19]}。图 4.1 是中断和异常管理的原理图。

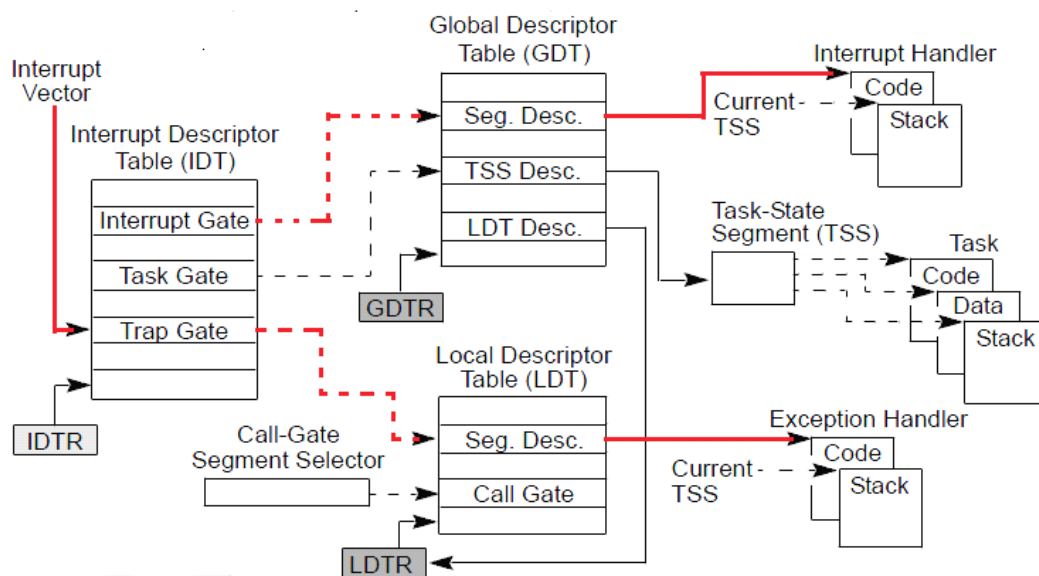


图 4.1 中断和异常管理

中断和异常就是事件 (events)，这些事件表明有需要处理器注意的特殊情况存在于系统，可能是处理器，或者当前执行程序的某个位置^[10]。中断和异常一般会导致执行过程从当前正在执行的程序强制转移到一个叫做中断处理程序或异常处理程序的特殊程序或任务入口处。

中断发生在程序执行过程的随机时间内，以响应从系统内部或外部发来的信号。例如系统的硬件会使用中断来处理外围设备的服务请求，软件也可以通过执行 `INT n` 指令来产生异常。处理器在执行指令的同时检测到了一个错误情况，例如被 0 除，异常就会发生，处理器会检测很多种错误情况，包括违反保护，页故障，和内部机器故障。除此之外，Intel Xeon 处理器还会在检测到内部硬件错误和总线错误时允许产生机器检查异常。

当收到中断或检测到异常时，当前运行的程序或任务会被挂起同时处理器会转去执行中断或异常处理程序。当处理程序执行完之后，处理器将恢复被中断的程序或任务。被中断程

序或任务的恢复应该会保持程序的连续性，除非不可能从异常情况中恢复或中断导致当前运行程序终止。

4.2 中断和异常向量

为了管理异常和中断，每个结构化定义的中断和异常情况都被分配了一个唯一的标识符，叫做向量号（vector）。处理器会使用分配给异常或中断的向量号作为索引，索引到中断描述符表（IDT）中，该表给出了中断或异常处理程序的入口点（如图 4.1）。

允许的向量号的范围为 0~255 号，其中 0~31 号由 IA32 and 64 架构预留给结构定义的异常和中断。不是所有 0~31 的向量号都有已经定义的功能，未分配的向量号是预留的，不能使用。32~255 号是用户可以定义的，并且不会被预留，这些向量号一般分配给外部的 I/O 设备，来让这些设备通过外部硬件中断机制给处理器发中断请求^{[19][20]}。

4.3 异常源

异常全部是由 IA32 and 64 架构结构化定义的，Intel Xeon-E5 处理器的异常向量号为 0~19，但是要除去向量号 2 和 15，向量号 2 为 NMI，15 为预留的。处理器收到的异常有 3 个来源^[20]：

- （1）处理器检测到的程序错误异常；
- （2）软件产生的异常；
- （3）机器检查异常。

4.3.1 程序错误异常

在系统中程序的执行过程中，如果处理器检测到程序错误它将会产生一个或多个异常。IA32 and 64 体系架构为每个处理器能检测到的异常定义了一个向量号，异常可以分为故障（faults），陷阱（traps），和终止（abort）。

4.3.2 软件产生的异常

INTO，INT 3 和 BOUND 这三条指令可以在软件中产生异常，它们分别对应向量号 3、4 和 5，这些指令允许在指令流中设置点来检查异常条件，例如 INT 3 会引起一个断点异常。

INT n 指令可以用来在软件中仿真异常，但是有一个限制：如果 INT n 提供了一个结构化

定义的异常的向量号，那么处理器将会产生对应向量号的中断，然后去执行异常处理程序。

4.3.3 机器检查异常

Intel Xeon-E5 处理器提供了内部和外部机器检查机制，来检查内部的芯片硬件和总线交易的操作过程。这些机制在实现上是独立的，当检测到一个机器检查错误时，处理器会发出一个机器检查的异常（向量号 18）并且返回一个错误码。

4.4 异常分类

异常被分类为故障（faults），陷阱（traps），和终止（abort）^[10]，分类情况取决于它们被上报的方式以及引起异常的指令是否会不失连续性的重启。

（1）故障指的是通常可以改正的异常，并且一旦改正了，程序会被允许不失连续性的重启。当一个故障被报告时，处理器会将状态还原到开始执行故障指令之前的机器状态。故障处理程序返回的地址（保存在 CS 和 EIP 寄存器中的内容）指向出故障的指令，而不是指向故障指令之后的指令。

（2）一个陷阱是在引起陷阱的指令执行之后直接被报出的异常，陷阱的产生通常会让处理器去执行陷阱处理程序。它能让程序或任务不失连续性地继续执行，陷阱处理程序返回的地址指向陷阱指令之后要执行的指令。

（3）终止异常不会总是报告引起异常的指令的位置，也不允许引起异常的程序或任务重启。终止一般用于报告很严重的错误，例如硬件错误和在系统表中出现不合法值。

4.5 中断源

对于 Intel Xeon-E5 处理器来说，中断的向量号为 2 和 32~255，其中 2 对应不可屏蔽中断（NMI）。处理器收到的中断有两个来源^[19]：

- （1）硬件产生的外部中断；
- （2）软件产生的中断。

4.5.1 外部中断

在 Intel Xeon-E5 处理器中，主要的中断引脚为 LINT[1:0]，它们连接在本地高级可编程中断控制器（APIC）上。当本地的 APIC 有效时，外部中断通过本地 APIC 接收到，然后本

地 APIC 通过自己的向量表（LVT）编程实现与处理器的异常或中断向量表（IDT）相关联。这些都是通过专门的 APIC 控制器实现的，本论文中不作深入研究。

处理器的本地 APIC 一般是连接在基于系统的 I/O APIC 上的，从 I/O APIC 的引脚上收到的外部中断可以通过系统总线送至本地 APIC 中，如图 4.2。I/O APIC 决定了中断的向量号，并把这个号码发送至本地 APIC，本项目中系统包含多个处理器，处理器可以将中断通过系统总线发送至其他的处理器^[20]。

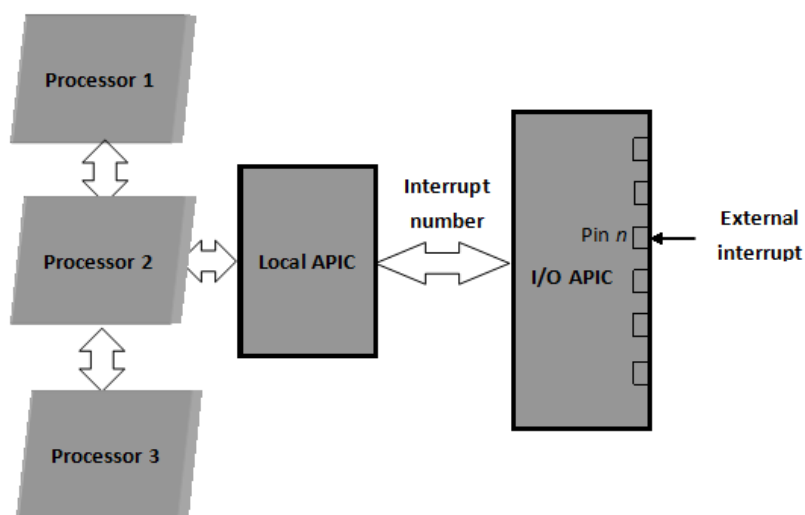


图 4.2 外部中断来源

如果本地 APIC 是全局的，它的引脚分别被配置成 INTR 和 NMI 引脚。触发 INTR 引脚将会发信号给处理器，告诉其外部中断发生了，处理器会从系统总线上读取外部中断控制器（8259A）提供的中断向量号。断定一个 NMI 引脚会发送一个不可屏蔽中断，其向量号为 2。

通过 INTR 引脚或通过本地 APIC 传递给处理器的任何外部中断都叫做可屏蔽硬件中断，通过 INTR 引脚传递的可屏蔽硬件中断包括所有 IA-32 定义的中断向量（0~255）；通过本地 APIC 传递的可屏蔽硬件中断包括 16~255 的中断向量。EFLAGS 寄存器中的 IF 标志位允许所有可屏蔽硬件中断作为一个组来被屏蔽。

4.5.2 软件产生的中断

在汇编指令中，INT n 指令使用中断向量号作为操作数来产生中断，例如，INT 35 指令实现了一个隐含的调用，调用中断 35 的中断处理程序，中断处理程序由用户定义。

0~255 的任何中断向量都可以用作这条指令的参数。但是，如果处理器的预先定义的 NMI 向量（向量号 2）没有使用，那么处理器做出的响应和已经定义的 NMI 中断的响应不同。如果 NMI 向量可以在该指令中使用，NMI 中断处理程序会被调用，但是处理器的 NMI 响应硬件不会被激活。EFLAGS 寄存器的 IF 标志位不能屏蔽 INT n 指令产生的中断。

4.6 同时发生的异常和中断的优先级

如果在一个指令边界上有多个异常或中断被触发，那么处理器会按照已经定义的顺序来响应它们，即按照优先级来处理。表 6-1 显示了异常和中断源的优先级。

表 6.1 同时发生的中断和异常的优先级^[10]

优先级	描述
1（最高）	来自硬件重启和机器检查 a. 重启 b. 机器检查
2	任务切换时的陷阱 a. 设置 TSS 的 T 标志位
3	外部硬件的中断信号 a. FLUSH b. STOPCLK c. SMI d. INIT
4	当前指令的上一条指令发生的陷阱 a. 断点 b. 调试陷阱异常
5	不可屏蔽中断 NMI
6	可屏蔽硬件中断
7	代码断点故障
8	预取下一条指令产生的故障 a. 超出代码段大小限制 b. 代码页故障
9	译解下一条指令产生的故障 a. 指令长度>15 字节 b. 无效操作数 c. 协处理器不可用
10（最低）	执行一条指令产生的故障 a. 溢出 b. 越界 c. 无效 TSS d. 指定段不存在 e. 堆栈故障 f. 数据页故障 g. 对齐检查错误 h. X87 浮点指令异常

4.7 程序或任务的继续执行

处理器通常在处理完中断和异常之后都会让原来的程序或任务继续执行。为了在处理完

异常或中断之后程序或任务能正常继续，所有的异常（除了 `aborts`）必须保证在指令边界处被上报，所有的中断也必须保证在指令边界处被处理^{[19][20]}。根据中断和异常的不同，程序或任务的继续执行也会不同：

（1）对于故障级别的异常，返回指令指针指向了引起故障的指令，所以，一个程序在处理完故障之后会重新执行引起异常的指令。这种情况常见于页故障异常（`page-fault exception, #PF`）：当一个程序要引用的操作数所在的页不在内存时，它就会产生一个 `#PF` 异常，然后异常处理程序会将该页加载到内存，并返回程序中重新执行引起故障的指令。处理器通过保存必需的寄存器和栈指针来使程序从故障指令之前的状态重启，以使程序不失连续性。

（2）对于陷阱级别的异常，返回指令指针指向了引起陷阱的指令的下一条指令。但是如果一条转移指令执行期间检测到了陷阱，那么返回指令指针指向这个转移目的地。例如，如果在执行 `JMP` 指令期间有陷阱异常被检测到，那么返回指令指针指向 `JMP` 指令的目的地，而不是 `JMP` 指令的下一条指令。所有的陷阱异常都允许程序不失连续性的重启，例如，溢出异常是一个陷阱异常，它是由检测 `EFLAGS.OF`（溢出）标志的 `INTO` 指令引起，那么返回指令指针指向该指令的下一条指令，陷阱异常处理程序会解决溢出条件。在从处理程序返回时，程序会接着执行 `INTO` 指令的下一条指令。

（3）终止级别异常不支持程序的可靠重启，异常处理程序只是被设计当终止异常发生时用来收集关于处理器状态的诊断信息，然后处理器会关闭应用程序和系统。

（4）中断强烈支持被中断的程序或任务不失连续性的重启，返回指令指针指向发生中断的指令边界的下一条要执行的指令。

4.8 中断描述符表和中断描述符

前面描述了异常和中断的来源，优先级，以及程序或任务在处理完中断和异常之后的继续执行，本节将重点描述异常中断描述符表和中断描述符。

4.8.1 中断描述符表

从图 4.1 可以看出，异常和中断的处理是通过中断或异常向量关联的中断描述符表是实现的。中断描述符表（`IDT`）将每个中断或异常向量和一个通道描述符关联起来，这个描述符会关联相应的异常或中断处理程序。和 `GDT`、`LDT` 一样，`IDT` 也是由 8-byte 描述符组成的队列，但是不像 `GDT`，`IDT` 第一个入口可能会包含一个描述符。为了产生一个到 `IDT` 的索引，处理器会按 8 的比例缩放异常或中断向量号，因为只有 256 个中断异常向量号，所以 `IDT` 只

需要包含少于 256 个描述符。中断描述符表中包含的描述符可以少于 256 个，因为只有那么可能发生的中断或异常向量号才有对应的描述符，而且所有空的描述符都应该置 0^[10]。

IDT 的基地址是 8-byte 边界对齐的，限制值加上基地址就能得到最后有效地址，因为 IDT 入口都是 8-byte 长，所以限制值应该为 $8n-1$ 。

IDT 可能驻留在线性地址空间的任何地方，处理器使用 IDTR 寄存器来定位 IDT，如图 4.3 所示。LIDT 和 SIDT 指令用来加载和存储 IDTR 寄存器的内容。LIDT 指令用来将内存操作数加载到 IDTR 中，而且 LIDT 只有在 CPL=0 时才可以执行，LIDT 通常只是在当一段 OS 初始化代码创建一个 IDT 时执行。SIDT 指令复制 IDTR 寄存器的基地址和限制值到内存中，任何优先权都能执行。

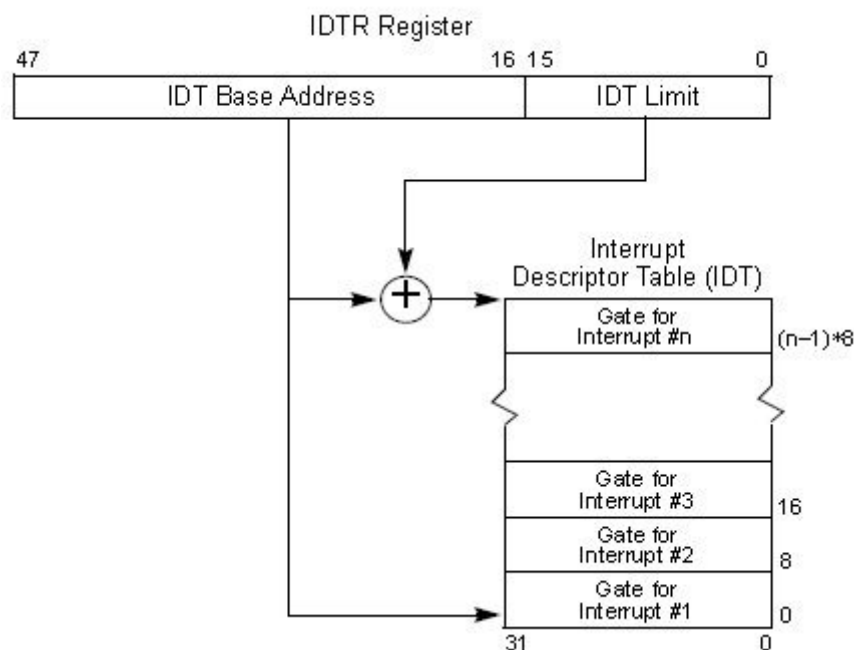


图 4.3 IDTR 和 IDT 的关系

4.8.2 IDT 描述符

IDT 中断描述符表中包含了 IDT，如图 4.1 所示，IDT 中可能包含下面三种中断描述符：

- (1) 任务通道 (task gate) 描述符
- (2) 中断通道 (interrupt gate) 描述符
- (3) 陷阱通道 (trap gate) 描述符

这些通道描述符我们习惯上称为通道^[10]。图 4.4 和图 4.5 中显示了任务通道，中断通道以及陷阱通道描述符的格式，它们大小都为 8byte。IDT 的任务通道和 GDT 或 LDT 的任务通道格式一样的，包含异常或中断处理程序的 TSS 的段选择子。陷阱和中断通道和调用通道相似，包含一个远程指针（段选择子和偏移量），处理器使用这个远程指针将程序执行过程转移到一

个异常或中断处理程序代码段。

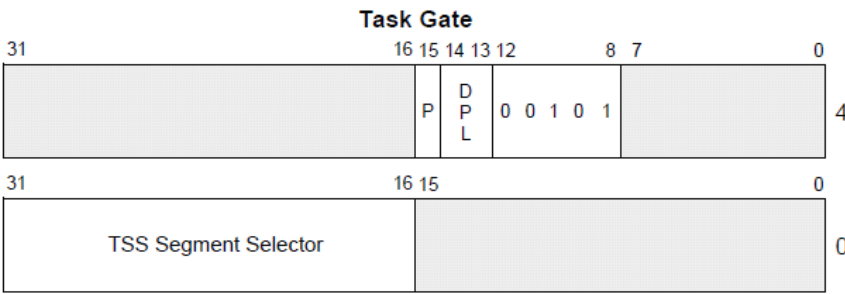


图 4.4 任务通道描述符的格式

其中需要说明的是，DPL 为描述符权限级，P 标志位表示其是否在段中。[12:8]字段为 00101B，表示这是一个系统段描述符，且为任务通道描述符（3.3.2 节）。

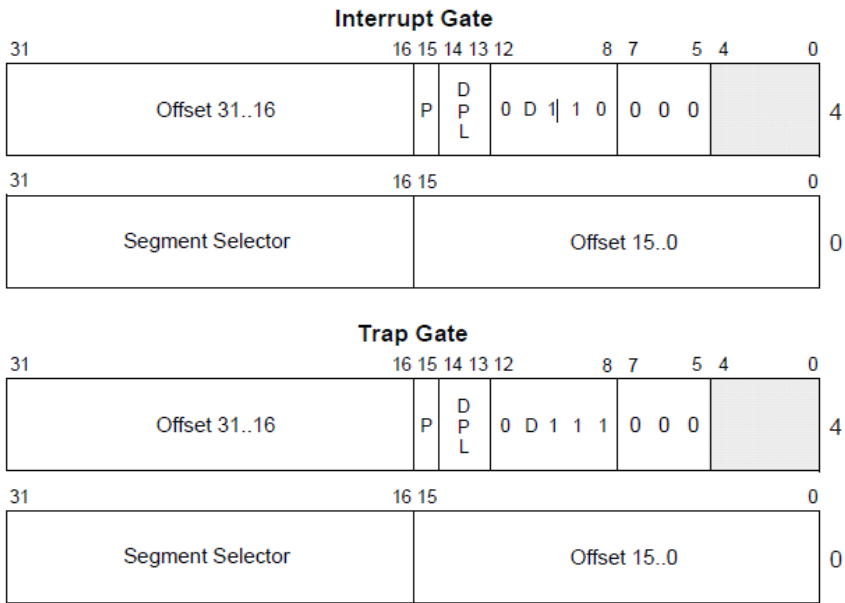


图 4.5 中断通道和陷阱通道描述符的格式

其中，段选择子为中断或异常处理程序的目标代码段的选择子，Offset[31:16]为处理程序入口的偏移量，D 标志位表示通道的大小。

4.9 异常和中断处理机制

当处理器在执行程序的过程中接收到中断信号或检测到异常时，除了特殊的终止异常外，一般情况下处理器都会转去执行对应的异常和中断处理程序。处理器会将某个异常或中断对应的中断向量作为 IDT 描述符的索引，如果 IDT 描述符是一个中断通道或陷阱通道，处理器会调用异常或中断处理程序；如果 IDT 描述符是一个任务通道，处理器会像正常使用 CALL 指令调用一个任务通道一样去执行一个任务切换的过程^{[10][21]}。

4.9.1 异常或中断处理程序调用过程

一个中断通道或陷阱通道会索引到当前执行任务上下文中的中断或异常处理程序，具体过程和分段机制是一样的。通道的段选择子指向可执行代码段的段描述符，代码段的描述符可能位于 GDT 或 LDT 中，通道描述符的偏移量指向异常或中断处理程序的开始处^[22]，如图 4.6 所示。

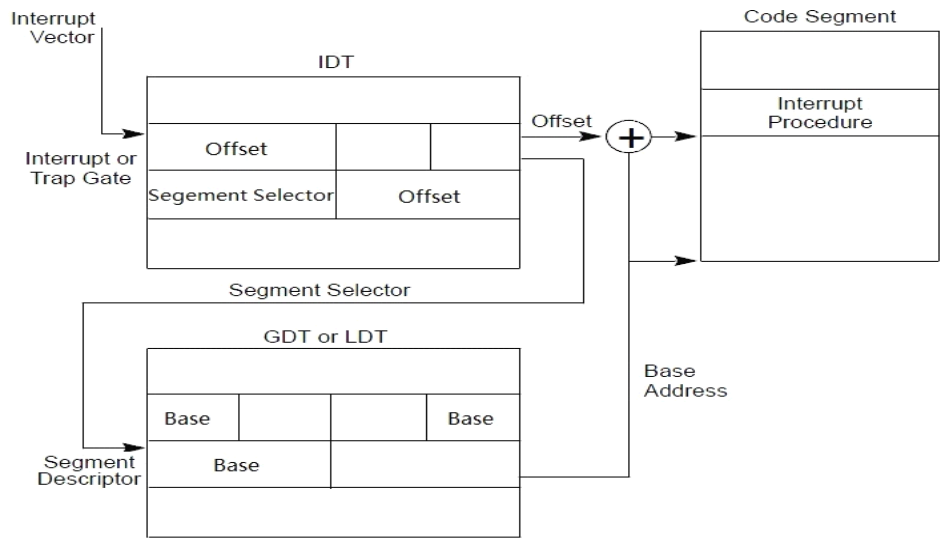


图 4.6 中断或异常处理程序的调用

我们知道一个可执行的任务必须包含代码段和堆栈段，当处理器收到异常或中断去调用异常或中断处理程序的时候，要进行对目标代码段的权限级检查，不同权限级之间的调用会造成堆栈切换问题^{[10][20]}。如果要执行的中断或异常处理程序有较高优先级，那么将会发生堆栈切换^[21-23]（如图 4.7）：

（1）从当前正在执行的任务的 TSS 中可以获得处理程序将要使用的堆栈的指针和段选择子，然后在这个新的堆栈中，处理器压入被中断的程序的堆栈段选择子和堆栈指针。

也就是说，中断或异常处理程序使用的堆栈是用来保存被中断的程序的堆栈段选择子和堆栈指针的。

（2）然后处理器会保存当前 EFLAGS、CS、EIP 寄存器的状态到新的堆栈中。

（3）如果异常响应过程产生了错误码，那么错误码会被压在新堆栈的 EIP 后面。

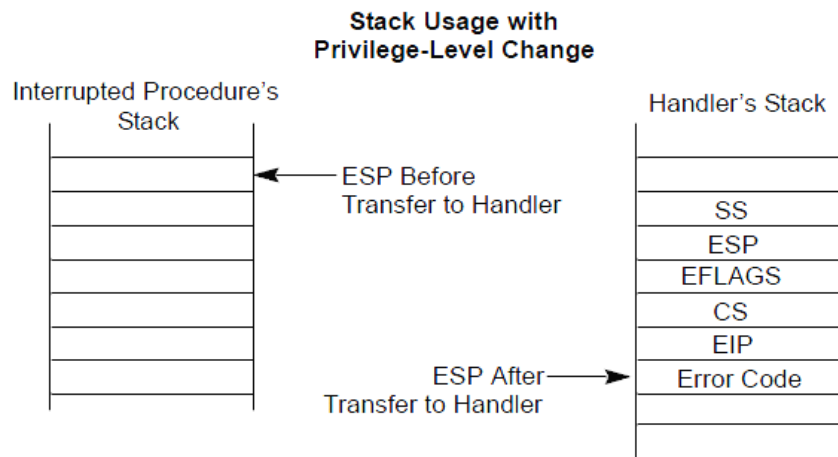


图 4.7 不同权限级会发生堆栈切换

如果要执行的中断或异常处理程序和被中断的程序同一优先级（图 4.8）：

- （1）处理器会保存当前的 EFLAGS、CS、EIP 寄存器到当前堆栈中。
- （2）如果产生了要保存的错误码，它将会放在 EIP 值后面。

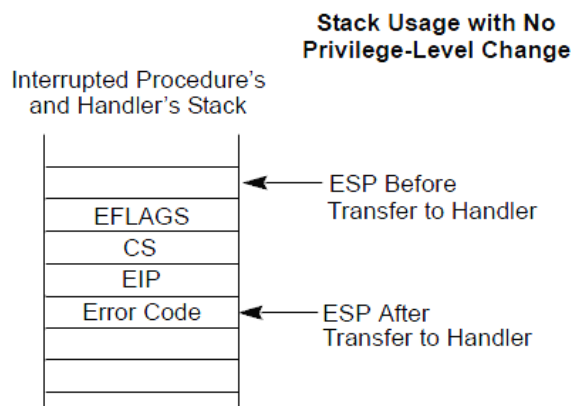


图 4.8 相同权限级不会堆栈切换

在异常或中断处理程序的最后要通过 IRET 指令使处理器执行过程从处理程序中返回到被中断的任务中，处理器会依次弹出堆栈段的内容并加载至对应的 EFLAGS、CS、EIP 等寄存器中，这样原来被中断的程序将会被继续执行。如果发生了堆栈切换，则 ESP 指针会返回到被中断程序的堆栈段位置^{[10][23]}。

4.9.2 中断任务的执行

在中断描述符表中，还有一种中断描述符为任务通道（描述符）。任务通道的使用一般并不是为了处理某个中断或异常，而只是处理器想通过它去执行某个特定的任务。当通过任务通道访问异常或中断处理程序时，会发生任务的切换过程（第五章会详细叙述）。通过单独的任务通道“处理”中断或异常有以下几个优点^{[10][21]}：

- (1) 被中断的程序或任务的全部执行环境会被自动保存下来。
- (2) 在处理异常或中断过程中，新的 TSS 可以使用最高权限级（0 级）的堆栈作为目标程序的堆栈。这样原来的 0 级堆栈在系统发生中断或异常情况而损坏时，就可以通过使用新的 0 级堆栈使系统继续运行。
- (3) 中断或异常处理程序可以放在相对独立的地址空间位置，这样可以将其与其他的任务隔离开。例如给其分配一个单独的 LDT。

任务通道索引的 TSS 描述符一定会放在 GDT 中，如图 4.9 所示，切换到处理程序的方式和任务切换过程一样。被中断程序的链接被加载至处理程序的 TSS 中，这样当执行过程从目标程序返回时，被中断程序的 TSS 也会被恢复。任务切换过程会在第五章任务管理中具体描述。

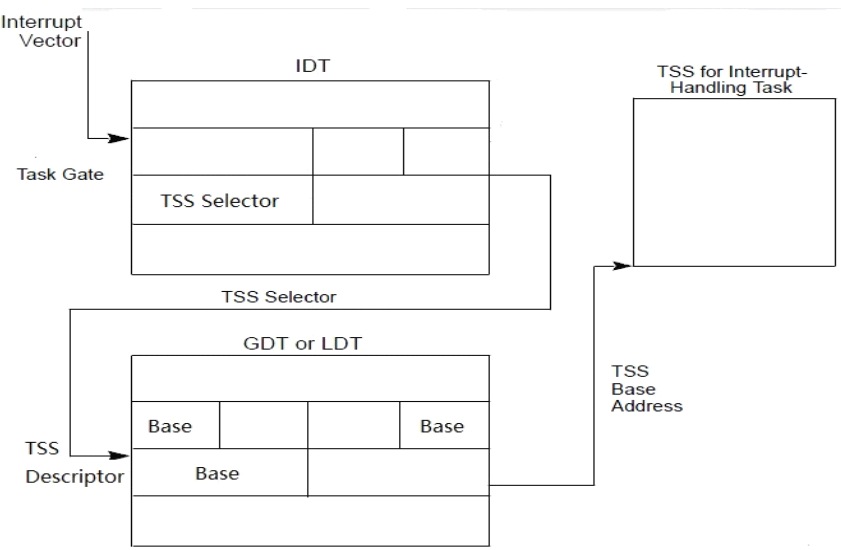


图 4.9 中断任务的执行

4.9.3 中断或异常处理机制的算法实现

前面几节中描述了中断和异常处理的具体机制，处理器要实现对一个中断或异常的响应，就必须在中断描述符表 IDT 中为该中断向量号分配对应的中断描述符。根据不同的中断或异常，我们希望处理器执行的中断或异常处理程序也是不一样，所以对应着不同的中断描述符。在驱动代码中，需要为系统中定义的每个中断或异常情况分配中断描述符，当处理器接收到中断或异常向量号时，就会去执行对应的处理程序^[22]。

根据 4.8.3 节中描述的中断和异常处理的机制以及关键的图 4.6 和 4.9，下面我们将用伪代码实现一个中断描述符表的创建，并新加入一个新的中断处理程序，为其分配对应的中断通道描述符，具体如下：

(1) 首先对中断通道描述符（图 4.5）的格式定义如下：

```
typedef struct {  
    UINT16    Offset_Low;  
    UINT16    SegmentSelector;  
    UINT16    Attributes;  
    UINT16    Offset_High;  
} INTERRUPT_GATE_DESCRIPTOR;
```

(2) 创建一个 IDT，并加入新的中断处理程序的描述符，InterruptHandler 为将要加入的中断处理程序：

```
void InitializeIDT (UINTN16 InterruptHandler)  
{  
    INTERRUPT_GATE_DESCRIPTOR *IDT_Entry;  
    INTERRUPT_GATE_DESCRIPTOR *IDT_Table;  
    UINT16 Code_Segment;  
    //创建 IDT，并读取用于放置目标程序的代码段位置  
    IDT_Table = 为 IDT_Table 分配内存;    // 即 256 * 8KB  
    Code_segment = 读取 CS 代码段寄存器中的值;  
    //读取当前 IDTR 的值, 通过它可以获得当前 IDT 的描述符的数量  
    //通过图 4.3 所示, 如果当前 IDT 中有 n 个描述符, 则 IDTR.Limit=8*n-1  
    IDTR = 获取当前 IDTR 的值;  
    IDT 中描述符的数量=(IDTR.Limit+1) / sizeof(INTERRUPT_GATE_DESCRIPTOR);  
    当前 IDT 中全部描述符的区域 = 从 IDTR.Base 到(IDTR.Limit+1)的区域;  
    IDT_Entry = IDT_Table;  
    将原先的 IDT 描述符搬到 IDT_Table 中, 起始位置为 IDT_Entry;  
    //此时 IDT_Table 的内存状态为图 4.10
```

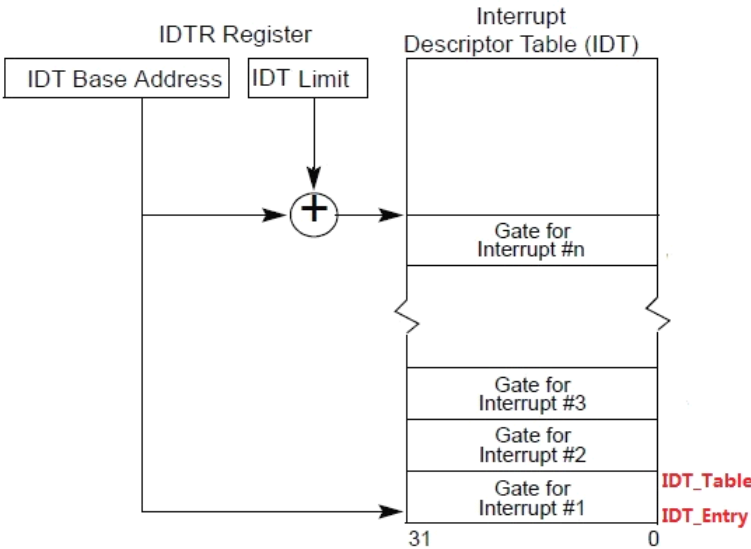



图 4.10 新创建的 IDT 内存区域

```
//为新的中断或异常处理程序分配中断通道描述符
for (Index = 0; Index < 256; Index ++){
    IDT_Entry[Index].SegmentSelector = Code_Segment;
    if (Index < 原先 IDT 中的描述符数量) continue;
    if (Index 是我想要设置的描述符)
    {
        IDT_Entry[Index].Offset_Low = InterruptHandler;
        IDT_Entry[Index].Offset_High = (InterruptHandler >> 16);
        .....;
    }
}
```

更新相应的寄存器;

```
}
```

4.10 本章小结

本章的内容主要讨论了处理器的中断和异常管理功能，重点研究了处理器如何管理中断和异常情况，以及当处理器收到或检测到中断和异常时如何处理。

在本项目中的 Intel Xeon 处理器架构中，系统将定义的所有异常和中断情况分配相应的中断向量，通过中断向量索引到中断描述符表 IDT 中，然后处理器通过 IDT 中的中断描述符找到对应的中断或异常处理程序并执行。在本章的 4.3~4.5 节中描述了中断和异常的来源及异

常的分类，4.6 节中叙述了同时发生的中断和异常的处理优先级。一般情况处理器在处理中断和异常情况后会回到被中断的程序中继续执行。本章的重点在于 4.8 节对中断和异常的处理机制，即通过 IDT 的索引去执行中断或异常处理程序。

在中断描述符表中包含三个通道描述符，对中断或异常处理程序的执行分为两种情况：一种是中断通道和陷阱通道索引的中断或异常处理程序，第二种是任务通道索引的目标任务。在本章中对两种情况都作了分析，并重点研究了前者的情况，详细分析了处理器从收到一个中断向量直到执行相应的中断或异常处理程序的过程。基于对过程算法的分析设计了创建一个 IDT 的伪代码实现，在代码中根据当前 IDTR 创建了 IDT 表，并对新加入的中断处理程序分配了相应的中断通道描述符。

第五章 任务管理

IA32 and 64 架构的任务管理机制仅当处理器运行在保护模式下时才能使用，本章将主要描述一个任务的组成结构，以及用于任务管理机制的重要的数据结构和寄存器，重点将研究 Intel Xeon 处理器任务管理功能的任务切换机制，并用伪代码片段描述任务切换机制的实现。本章所描述的任务管理机制主要针对 32 比特任务和 32 比特 TSS 结构。

5.1 任务管理综述

一个任务对于处理器来说就是一个工作单元，在这个工作单元里处理器可能调度，执行或者挂起。一个任务可以用来执行一段程序，一个进程，一个操作系统程序，或者一个中断或异常处理程序等等。在上一章中已经描述了用一个任务切换执行中断或异常处理程序的机制，当该处理程序通过任务通道描述符索引时，就可以通过任务切换实现中断的处理^[10]。

一个任务主要由任务状态段 TSS 指定，TSS 中包含了确定该任务执行空间和执行状态的信息。IA32 and 64 架构的任务管理机制提供了如下功能：保存一个任务的状态信息，调度任务执行，和切换不同的任务。处理器可以通过 TSS 段描述符以及 TSS 段选择子管理一个任务，例如通过保存 TSS 中的信息可以保存一个任务的状态，或通过任务通道描述符索引并加载 TSS 可以实现任务的切换^[23]。

用于任务管理的 TSS 段选择子可以通过 CALL 指令或 JMP 指令专门给出，也可以通过 IDT 中的任务通道描述符中给出。处理器任务管理机制的原理图为图 5.1 所示。

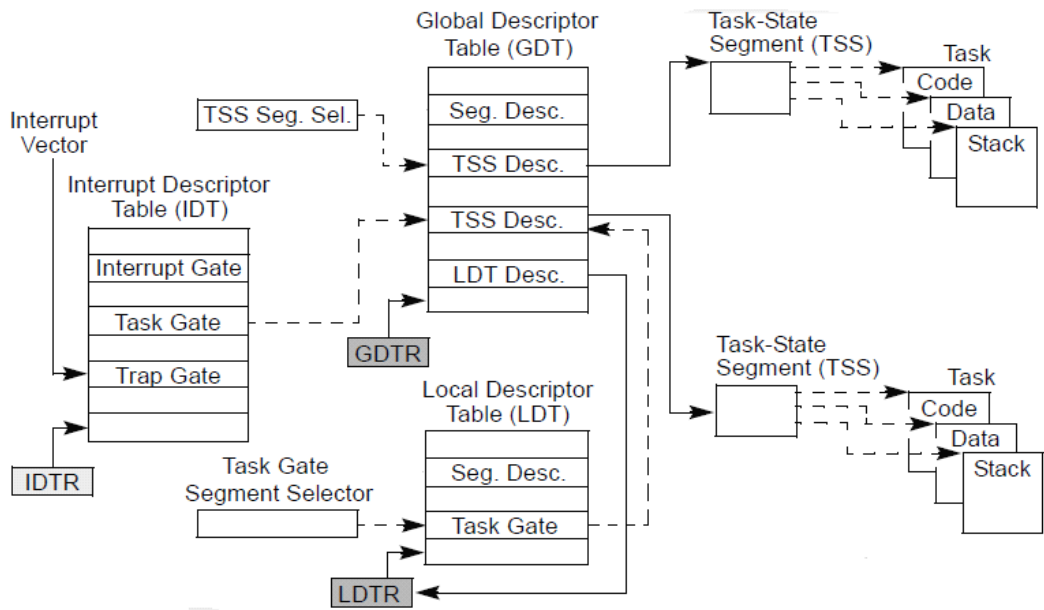


图 5.1 任务管理机制原理图

5.1.1 任务结构

一个任务由两个部分组成：任务执行空间（task execution space）和任务状态段（task-state segment），如图 5.2 所示。任务执行空间包括代码段，堆栈段和一个或多个数据段，如果操作系统使用了优先级保护机制，那么任务执行空间还要为每个优先级提供一个单独的堆栈^{[10][24]}。

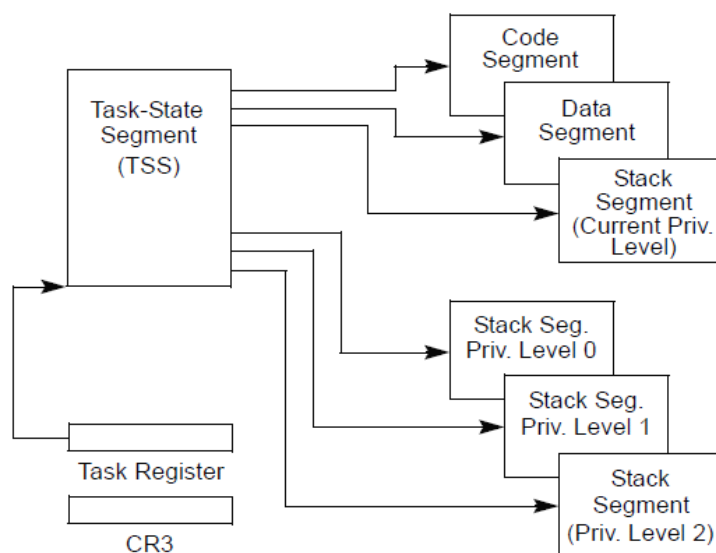


图 5.2 一个任务的结构

TSS 中包含了多个寄存器，它们用来指定组成任务执行空间的段，TSS 中还存储任务状态信息，在多任务系统中，TSS 也提供任务链接机制。所以，一个任务由 TSS 和与 TSS 对应的段选择子来确定，当一个任务被加载到处理器执行的时候，其对应的段选择子，基地址，段最大限和 TSS 段描述符属性都会被加载进任务寄存器。如果任务中使用分页，则任务使用的分页路径的基址要被加载进控制寄存器 CR3^[24]。

5.1.2 任务的状态信息

下列项定义了当前正在执行的任务的状态信息，它们基本全部被包含在 TSS 中：

(1) 任务的当前执行空间的位置，由段寄存器 CS、DS、SS、ES、FS 和 GS 中的段选择子定义

(2) 通用寄存器状态

(3) EFLAGS 寄存器状态

(4) EIP 寄存器状态

(5) 控制寄存器 CR3 状态

(6) 任务寄存器 TR 状态

(7) LDTR 寄存器状态

(8) I/O 映射基地址和 I/O 映射 (TSS 中)

(9) 堆栈指针, 指向优先级 0, 1, 2 堆栈 (TSS 中)

(10) 到之前已执行过的任务的链接 (TSS 中)

在调度执行任务之前, 除了任务寄存器状态之外, 所有的项都要被包含在任务的 TSS 中。还有要注意的是, 不是全部 LDTR 的内容都会被包含在 TSS 中, 只有 LDT 的段选择子会。

5.2 调度任务的执行

当处理器需要执行一个任务时, 可能会通过如下途径调度其执行^{[10][25]}:

- (1) 用 CALL 指令明确调用一个任务;
- (2) 用 JMP 指令明确跳转到一个任务;
- (3) 隐含调用一个中断处理程序的任务;
- (4) 隐含调用一个异常处理程序的任务;
- (5) 当 EFLAGS 寄存器中的 NT 标志位被置 1 的返回操作。

以上这些方法都是通过使用段选择子来指定要调度的任务实现的, 段选择子指向任务通道或 TSS 段描述符从而确定要调度的任务的位置。当使用 CALL 或 JMP 指令调度执行一个任务时, 指令中的段选择子可以直接选中 TSS, 或者间接使用包含段选择子的任务通道来选中 TSS。当通过处理中断或异常的方法调度执行一个任务时, 中断或异常处理程序的 IDT 入口必须被包含在任务通道中, 此任务通道包含了中断或异常处理程序的 TSS 的选择子^{[10][26]}。

当一个任务被调度执行的时候, 在当前任务和被调度任务之间就会发生任务切换。具体过程如下:

(1) 将当前正在执行的任务的执行环境 (通常叫做任务上下文, context) 保存在它的 TSS 中, 然后此任务的执行会被挂起;

(2) 将被调度执行的任务的上下文加载至处理器, 然后任务的执行以最新加载到 EIP 寄存器的指令开始。如果被调度任务从未执行过, 那么 EIP 将指向该任务的第一条指令位置; 否则 EIP 将指向上一次被调用时最后执行的指令的下一条指令。

(3) 如果当前正在执行的任务调用了正被调度的任务, 主调任务的 TSS 段选择子会被保存至被调任务的 TSS 中, 以此来提供返回主调任务的链接。

注意对于所有的 IA32 处理器不能循环递归, 即一个任务不能调用或跳转到自己本身。

在上一章的中断和异常处理机制中已经描述过, 处理器可以使用任务切换来完成中断和

异常的处理。这里，处理器会执行一个任务切换去执行中断或异常处理程序，并且在从处理程序的任务返回时会自动切换回被中断的任务^[27]。

在 4.9.1 节中断处理程序任务的执行中已经介绍过，可以将中断或异常处理程序放在相对独立的地址空间位置（LDT），这样每个任务都能有不同的逻辑地址到物理地址的映射。在任务切换时，用于分页的页路径基地址寄存器 CR3 需要重新被加载，这样每个任务都会有自己的一组页表。通过这种机制可以保证各个任务之间互不干扰。

5.3 用于任务管理的数据结构

就像中断和异常管理机制一样，任务管理机制也定义数据结构用来处理任务相关的操作，下面 5 个数据结构是由系统架构定义的：

- (1) 任务状态段（TSS）
- (2) 任务通道描述符（task-gate descriptor）
- (3) 任务状态段描述符
- (4) 任务寄存器（TR）
- (5) EFLAGS 寄存器的 NT 标志位

在保护模式的操作下，每个任务至少要创建一个 TSS 和一个 TSS 描述符，并且 TSS 的段选择子必须要被加载至 TR 中。下面几个小节将重点描述这些数据结构是怎样用于任务管理的。

5.3.1 任务状态段

任务状态段是任务管理中最重要数据结构，它包含了一个任务所有的状态信息，通过它可以保存一个任务状态，调度一个任务的执行和任务切换。图 5.3 显示了一个 TSS 的格式，一个 TSS 的字段被分成两个主要的类别：动态域和静态域^{[10][27]}。

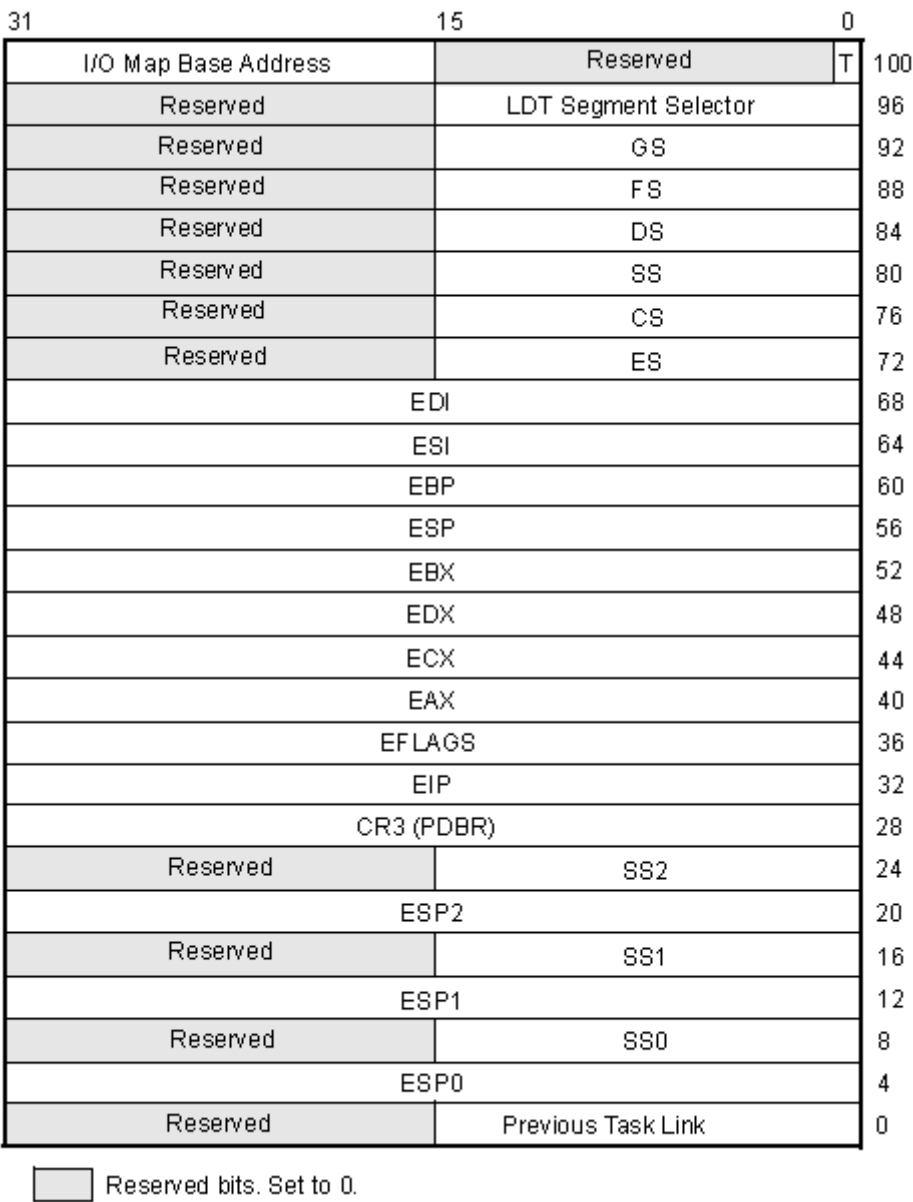


图 5.3 任务状态段的格式

在任务切换过程中当一个任务被挂起时处理器就会更新动态域，动态域包括下面的内容：

- （1）通用寄存器域，是指在任务切换发生之前的 EAX、ECX、EDX、EBX、ESP、EBP、ESI 和 EDI 寄存器的状态；
- （2）段选择子域，段选择子都会被保存在任务切换之前的 ES、CS、SS、DS、FS、和 GS 寄存器中；
- （3）EFLAGS 寄存器域，是指任务切换之前的 EFLAGS 寄存器的状态；
- （4）EIP（指令指针寄存器）域，是指任务切换之前的 EIP 指令指针寄存器的状态；
- （5）前一任务链接域，有时也叫后退链接域。该域包含了前一任务（即主调任务）的 TSS 的段选择子，当一个调用和中断情况导致任务切换时该域就会被更新。通过使用该域，处理器在执行 IRET 指令之后就可以返回到之前的主调任务中^[28]。

对于 TSS 的静态域，处理器通常只会读取其状态而不会改变它们的值，这些域是在一个任务被创建的时候就被设置的，如下是静态域内容：

- (1) LDT 段选择子域，包含该任务的 LDT 段选择子。
- (2) CR3 控制寄存器域，包含了该任务使用的页路径表的物理基地址，所以 CR3 一般也叫做页路径基址寄存器（PDBR）。
- (3) 权限级 0，1 和 2 的堆栈指针域，这些堆栈指针包含了一个逻辑地址，该逻辑地址由堆栈段（SS0，SS1，SS2）的段选择子以及堆栈偏移量（ESP0，ESP1，ESP2）组成。对于特定的任务来说堆栈指针域是静态的，但是在任务切换时如果发生任务切换则 SS 和 ESP 也会相应改变。
- (4) T（调试通道）标志位（第 100 字节，bit0 位），当该位置 1 时，处理器会在任务切换时触发一个调试异常。
- (5) I/O 映射基地址，包含一个相对于 TSS 基地址的 16bit 偏移量值。

如果在任务切换过程中使用了分页机制，那么如下几点需要注意：

- (1) 避免将页面边界放在一个任务的 TSS 最开始的 104 字节中。因为在任务切换时，处理器会读写每个 TSS 的最开始 104 个字节，而且从第一个字节开始 TSS 是连续的物理内存地址。如果页面边界出现在这个区域处理器可能不会正确执行地址转换过程。
- (2) 对应前一任务的 TSS，当前任务 TSS 和描述符表入口的页都应该被标记为读或写。
- (3) 如果在开始任务切换之前包含任务管理数据结构的页已经存在于内存中，那么切换速度会加快。

5.3.2 TSS 描述符

TSS 像所有其他段一样，也是通过段描述符确定的，但是 TSS 描述符只能放在 GDT 中，不能放在 LDT 和 IDT 中。图 5.4 为一个 TSS 描述符的格式。

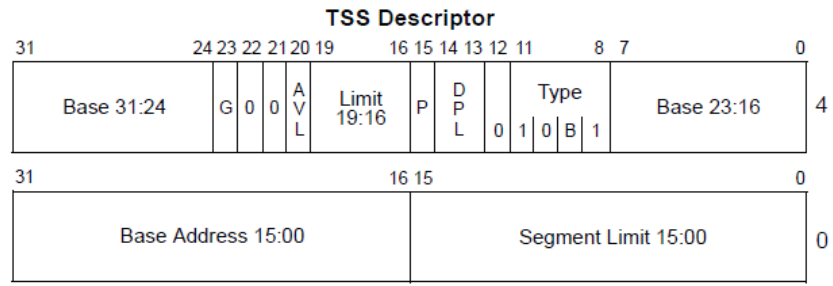


图 5.4 TSS 段描述符的格式

type 字段的 B 标志位用来表明任务是否正忙，一个忙任务是指当前正在被运行，或者已被挂起。例如 type 字段为 1001B 表示这是一个闲置的任务，1011B 表示这是一个忙任务。前

面已经描述任务是不能循环切换的，所以处理器可以使用 B 标志位来检测是否在尝试切换到执行过程已被中断的任务，为了保证一个任务只有一个 B 标志位，每个 TSS 应该只有一个 TSS 描述符^[29]。

任何执行过程或程序，如果其 CPL 在数字上等于或小于 TSS 描述符的 CPL，都可以访问 TSS 描述符，也就是说都可以调度任务。在大多数的系统中，TSS 段描述符的 DPL 字段都被设置为小于 3，这样只有有特权的软件才可以执行任务切换。

5.3.3 任务寄存器

任务管理机制中还有一个重要的寄存器，任务寄存器 TR，当任务切换开始时，处理器用它来索引当前任务的任务状态段的信息。任务寄存器包括了当前任务 TSS 的 16 比特段选择子和整个段描述符（32 比特基地址，16 比特段大小限制和段描述符属性），这些信息都是从 GDT 中的 TSS 描述符拷贝到 TR 中的，下图 5.5 为一个 TR 的格式：

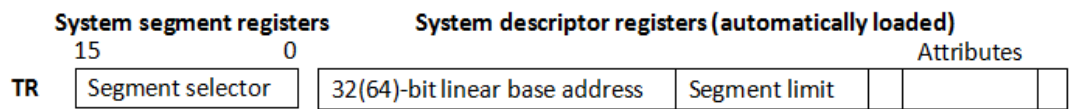


图 5.5 任务寄存器 TR 的格式

任务寄存器有一个软件可见部分和软件不可见部分，如图 5.6 所示，段选择子处于可见部分，它指向 GDT 中的段描述符。不可见部分主要用来缓存 TSS 的段描述符，这样能使任务执行更高效。加载和读取任务寄存器的可见部分是通过 LTR 指令和 STR 指令完成的。LTR 用来加载段选择子到 TR 中，然后 TSS 描述符的信息会被自动加载到 TR 中。STR 用来将可见部分保存进通用寄存器或内存，以便确定当前正在执行的任务^{[10][27]}。

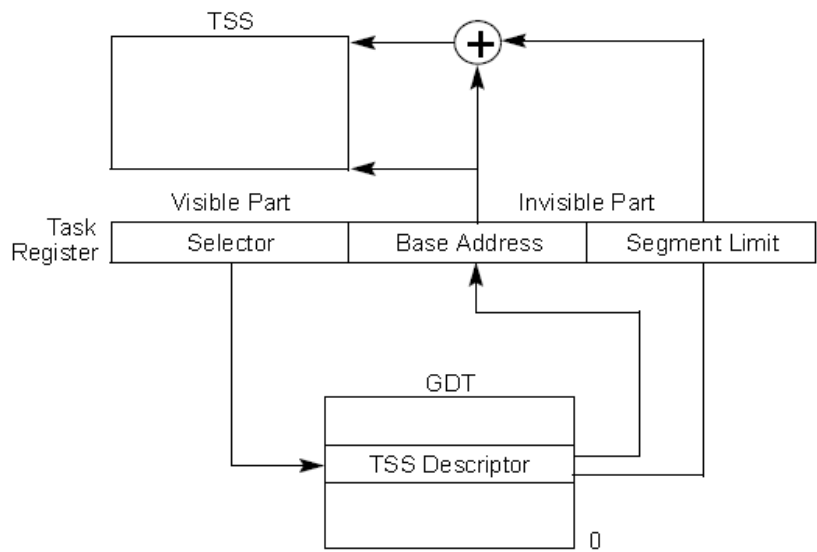


图 5.6 任务寄存器的使用

5.3.4 任务通道描述符

处理器对于任务切换的数据结构还有一个任务通道描述符（也叫任务通道，Task Gate），任务通道描述符提供了一种间接地，受保护地引用任务的方法，它可以放在 GDT，LDT 或者 IDT 中。但是任务通道描述符的 TSS 段选择子都会指向 GDT 中的一个 TSS 描述符。任务通道描述符的 DPL（descriptor privilege level）控制对 TSS 描述符的访问^[27]。任务通道描述符的格式如图 5.7 所示。

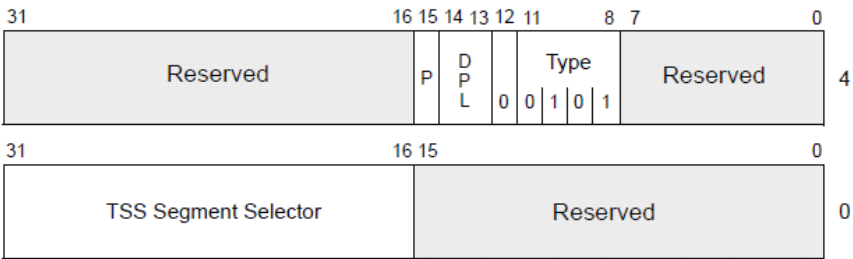


图 5.7 任务通道描述符

综上所述，一个任务可以通过任务通道描述符访问，也可以直接通过 TSS 段描述符访问。不管哪种方式，任务切换时对任务的访问要满足以下的必要条件：

- （1）需要该任务只有一个 B 标志位。这是因为表示一个任务是否忙的 B 标志位被保存在 TSS 描述符中，而且每个任务只能有一个 TSS 段描述符。但是需要注意多个任务通道可以索引到同一个 TSS 描述符。
- （2）访问任务时需要提供一个可选的途径。任务通道描述符就满足这个条件，因为任务通道可以放在 LDT 中，它们也可以有不同于 TSS 描述符的 DPL 的 DPL。这样如果一个程序没有足够的权限访问 GDT 中的 TSS 描述符的时候，就可以通过有更多 DPL 的任务通道描述符访问该任务。任务通道描述符给了系统访问指定任务的更高的界限。
- （3）需要在一个独立的任务中就可以处理一个中断或异常。任务通道描述符可以存放在中段描述符表 IDT 中，这就允许系统通过处理程序任务来处理中断和异常情况。当一个中断或异常向量指向一个中断通道的时候，处理就可以切换到指定的处理程序任务。

5.4 任务切换机制及算法实现

前面两节描述了调度一个任务执行的过程，以及用于任务管理的重要数据结构，这一节将重点研究处理器如何实现多个任务之间切换，并通过代码片段简要阐述为每个逻辑处理器创建一个任务状态段的过程。

5.4.1 任务切换机制

处理器可以通过以下四种情况执行任务的切换^[10]:

- (1) 当前程序或任务执行一个 JMP 或 CALL 指令, 跳转到 GDT 中的 TSS 描述符;
- (2) 当前程序或任务执行一个 JMP 或 CALL 指令, 跳转到 GDT 或 LDT 的一个任务通道描述符;
- (3) 当前中断或异常向量号指向中断描述符表 IDT 中的一个任务通道描述符;
- (4) 当前任务在 NT 标志位置 1 时执行 IRET 指令。

JMP, CALL, IRET 指令和中断或异常, 这些都是可以实现程序切换的机制, 一个 TSS 描述符或任务通道的引用或 NT 标志位的状态决定了是否发生任务切换。处理器在切换新任务时会执行如下操作:

(1) 新任务的 TSS 段选择子作为 JMP 或 CALL 指令的操作数, TSS 段选择子可以从任务通道中获得, 或者当执行 IRET 指令时从前一个任务链接字段获得。

(2) 检查当前任务是否允许切换到新任务, 当前任务的 CPL 和新任务段选择子的 RPL 必须小于等于将要被引用的 TSS 描述符或任务通道的 DPL。

异常, 中断和 IRET 指令允许不检查目标任务通道或 TSS 描述符的 DPL, 但是对于 INTn 指令, DPL 会被检查。

(3) 检查新任务的 TSS 描述符是否被标记为 present 并且有一个有效限制值。

(4) 检查新任务是否有效 (CALL, JMP, 异常或中断情况) 或忙碌 (IRET 指令)。

(5) 检查当前 TSS, 新 TSS, 和所有在任务切换中使用的段描述符都已经被加载至系统内存。

(6) 如果任务切换由 JMP 或 IRET 指令启动, 处理器会清除当前任务 TSS 描述符的 B 标志位; 如果由 CALL 指令, 异常或中断启动, B 标志位还是会被置 1。

(7) 如果任务切换由 IRET 指令启动, 处理器会清除 NT 标志位; 如果由 CALL 或 JMP 指令, 异常或中断启动, NT 标志位不会被改变。

(8) 保存当前任务的状态到当前任务的 TSS 中, 处理器从任务寄存器中找到当前 TSS 的基地址, 然后复制下面寄存器的状态到当前 TSS: 所有通用寄存器, 段寄存器中的段选择子, 暂时保存的 EFLAGS 寄存器和 EIP 寄存器的复制值。

(9) 如果任务切换由 CALL 指令, 异常或中断发起, 处理器会设置新任务的 NT 标志位为 1; 如果是由 IRET 或 JMP 指令发起, NT 标志位将会反映新任务 EFLAGS 的 NT 状态。

(10) 如果任务切换由 CALL 指令, JMP 指令, 异常或中断发起, 处理器会将新任务 TSS

描述符的 B 标志位置 1；如果由 IRET 发起，B 标志位不变。

(11) 加载新任务 TSS 的段选择子和描述符到任务寄存器中。

(12) 加载 TSS 状态到处理器：LDTR, CR3, EFLAGS, EIP, 通用寄存器和段选择子。

(13) 和段选择子相关的描述符被加载并获得权限，任务加载和取得权限过程中产生的错误都会发生在新任务的环境下。

(14) 开始执行新任务。

当一个任务切换成功执行时，当前正在执行的任务的状态总是会被保存。当目标任务执行完回到主调任务时，执行过程会从已保存的 EIP 寄存器值所指向的指令开始，并且当该主调任务被挂起时保存的寄存器组的值会重新被恢复对应的寄存器中^[26]。

当切换任务时，新任务的权限级不会从被挂起的任务那里继承，而是会以 CS 寄存器的 CPL 字段中指定的权限级执行。因为一个任务是被单独的地址空间和 TSS 隔离开的，而且因为权限规则控制了 TSS 的访问，所以软件不需要在任务切换时专门进行权限级检查。

5.4.2 任务切换机制实现

通过以上实现任务切换的步骤来看，任务切换机制和中断或异常处理程序的执行过程类似，不同的是对 TSS 任务状态段的访问和切换，如下的程序片段是创建 GDT 中的 1 个 TSS 段描述符的伪代码实现。

```
for (Index = 0; Index < 全部逻辑处理器的数量; Index++) {  
    调用 CopyMem 函数从当前的 GDT 偏移量位置开始创建一个 TSS 的内存空间(104B);  
    //填充 TSS 段描述符的内容  
    Tss_Base = (UINTN)(GdtTssTables + GdtTssTableSize * Index + Gdtr.Limit + 1);  
    TSS_Descriptor = (IA32_SEGMENT_DESCRIPTOR *)(Tss_Base) - 2;  
    //接下来给 TSS 段描述符的基地址重新赋值  
    TSS_Descriptor.BaseLow = (UINT16)Tss_Base;  
    TSS_Descriptor.BaseMid = (UINT8)(Tss_Base >> 16);  
    TSS_Descriptor.BaseHigh = (UINT8)(Tss_Base >> 24);  
    Tss_Base += 104;    //104 字节为一个 TSS 的大小  
    TSS_Descriptor++;  
    .....  
}
```

以上代码实现了为每个逻辑处理器分配一个任务状态段 TSS 和相应的 TSS 段描述符的过程。其他的寄存器的操作以及关键数据结构的定义和中断和异常管理机制的操作类似，这里不再深入研究。

5.5 本章小结

本章的内容描述了 IA32 and 64 架构处理器的任务管理功能，主要阐述了处理器的一个任务的组成部分，需要管理的数据，并重点研究了处理器任务管理时使用的关键数据结构以及如何使用这些数据结构执行任务管理功能。

在本项目中的 Intel Xeon 处理器架构中，系统中每个任务都包含了丰富的信息，主要包括任务执行空间和任务状态段 TSS 组成，任务执行空间主要包括代码段，堆栈段和一个或多个数据段，它定义了一个任务执行所需要的所有数据。任务状态段 TSS 是一个任务最重要的数据结构，它指定了一个任务的执行空间以及一个任务在运行时的状态空间。调度系统中一个任务使之开始执行可以通过 4 种方式实现，在 5.2 中分别对一个任务的启动过程作了简要的概述。

在接下来的 5.3 节中重点描述了实现任务管理功能所必须的数据结构。其中 TSS 是一个任务最重要的数据结构，它包含了或指定了一个任务所有的状态信息，通过它可以保存一个任务状态，调度一个任务的执行和任务切换。每个任务的 TSS 都包含动态域和静态域，在任务切换时处理器会更新 TSS 的动态域，对于静态域只会读取其内容而不会改变它们。在上一章的中断和异常管理中已经阐述了有关任务通道描述符的内容，但是运用任务通道描述符访问一个任务的状态段有很多优点，例如可以提升权限级，但是另一方面会减慢任务切换的速度。任务寄存器 TR 可以被处理器用来获得当前正在执行的任务的信息。

如果任务切换发生在不同权限级的程序之间，可能就会发生堆栈的切换，这需要进行目标任务状态段的权限级检查。对于任务切换还有几个限制和必须条件，这些在使用任务管理时都要注意。

在最后一节中重点研究了任务管理功能中最重要的任务切换机制，详细分析了任务切换所遵循的算法步骤，最后通过伪代码片段的方式简要阐述了为每个逻辑处理器创建任务状态段的过程。

第六章 多处理器机制

Intel IA32 and 64 架构提供了管理和提升连接在同一系统总线的多处理器的性能的机制，这就是多处理器机制^{[10][29]}。本论文中的 Xeon-E5 处理器也是采用多处理器机制，多处理器机制主要有以下内容：

(1) 保持系统内存的联接——当两个以上的处理器同时尝试访问同一系统内存地址时，一些处理器之间的通信机制或内存访问协议将会用来保持数据的联接性。

(2) 保持缓存的一致性——当一个处理器访问在另一个处理器上缓存的数据时，前者一定不能随意修改数据。如果它修改了数据，那么所有其他处理器都将访问修改过的数据。

(3) 允许预先排好写入内存的顺序 (memory ordering)^[30]——在很多情况下，内存写操作都要精确地遵循编程好的顺序。

(4) 在一组处理器之间处理多个中断应用请求——当多个处理器并行执行任务时，需要有一个集中的机制来接收中断并分配中断给多个处理器。

(5) 可以通过开发多线程和多进程来提升系统性能。

本章的四个小节将对上述内容作实现上的具体阐述，6.1 节上锁的原子操作保证了内存的联接性，6.2 节内存有序化阐述内存的访问怎样实现有序，6.3 节是多处理器的初始化，6.4 节多处理器的管理实现多处理器并行执行任务的协调机制。对于多处理器的驱动来说，实现多处理器的初始化以及并行执行程序是最关键的部分。

6.1 上锁的原子操作

原子操作是指不会被其他调度机制打断的操作，是不可分割的，在执行完毕之前不会被任何其它任务或事件中断，一旦开始就一直运行到结束^[31]。上锁的原子操作对于管理系统内存的共享数据结构非常有用，当两个或两个以上处理器同时尝试修改相同的字段或标志位时，就要保证原子操作。执行原子操作的机制有三种：

- (1) 有保证的原子操作，对于 Intel Xeon 处理器来说，基本的内存读写操作都是原子地；
- (2) 通过 LOCK#信号和 LOCK 指令前缀对总线上锁；
- (3) 缓存一致性协议。

6.1.1 给总线上锁

当遇到关键的内存操作时，处理器就会自动断言一个 LOCK#信号来对系统总线上锁。当这个输出信号被断言时，其他处理器对该总线的控制请求将被阻塞。给总线上锁有两种情况，第一种是自动上锁，另一种是软件控制总线上锁^[10]。

自动上锁主要有几种情况：

(1) 执行 XCHG 指令索引内存。

(2) 设置 TSS 描述符的 B 标志位。为了保证两个处理器不会切换到相同的任务中，处理器在检测和设置 B 标志位时会遵循锁语义。

(3) 更新段描述符。当加载段描述符时，处理器会设置 `accessed` 标志位，在这个操作过程中必须保证该描述符不会被其他处理器修改。

(4) 更新页路径和页表入口。当更新页路径和页表入口时，处理器会设置 `accessed` 和 `dirty` 标志位。

(5) 确认中断。中断发生后，中断控制器会通过数据总线将中断向量发送给处理器，这个过程必须保证没有其他数据出现在数据总线上。

为了明确 LOCK 语义，代码中我们可以在一些修改内存数据的指令前面加 LOCK 前缀，这样就能保证该指令的操作在执行时总线是上锁的。上锁的指令可以用来同步数据：在一个处理器上写入，在另一个处理器上读取。如某个处理器申请 `MPlock` 的函数中

```
lock cmpxchg [ecx], edx ;execution of cmpxchg instruction following LOCK semantics
```

一个总线锁并不会受内存字段是否对齐的影响，但是推荐总线锁操作应该是自然对齐的，这样可以提升系统性能。

6.2 内存有序化

内存有序化（memory ordering）指的是处理器通过系统总线分发系统内存读和写操作的顺序，内存有序化在 Intel Pentium 4 处理器中引入^[31]。内存有序化能够允许性能提升的操作，比如允许读操作在缓存好的写操作之前执行，这样主要能提升指令执行速度，还能在多处理器系统中保持内存一致性。

Pentium 和 Intel486 处理器遵循处理器排序的内存模型，读和写操作总是按照处理器预先排好的顺序出现在系统总线上。而对于 Intel Xeon 处理器来说，它会保证对共享变量的访问要明确地服从通过软件排好的顺序，这些可以通过使用正确的上锁或序列化的操作步骤。

图 6.1 是一个多处理器内存有序化的例子。假设系统有三个处理器且每个处理器执行了三个写操作，每个写操作都是对三个相同的内存位置（A，B，C）。对于单独的处理器来说，它们的写操作是按照软件排好序的顺序执行的。但是由于存在总线仲裁和其他内存访问的机制，每次各自的代码序列在处理器上执行的时候，三个处理器对单独的内存位置写操作的顺序可能会不同，最后 A，B，C 位置的值可能会因为每次写顺序不同而不同。

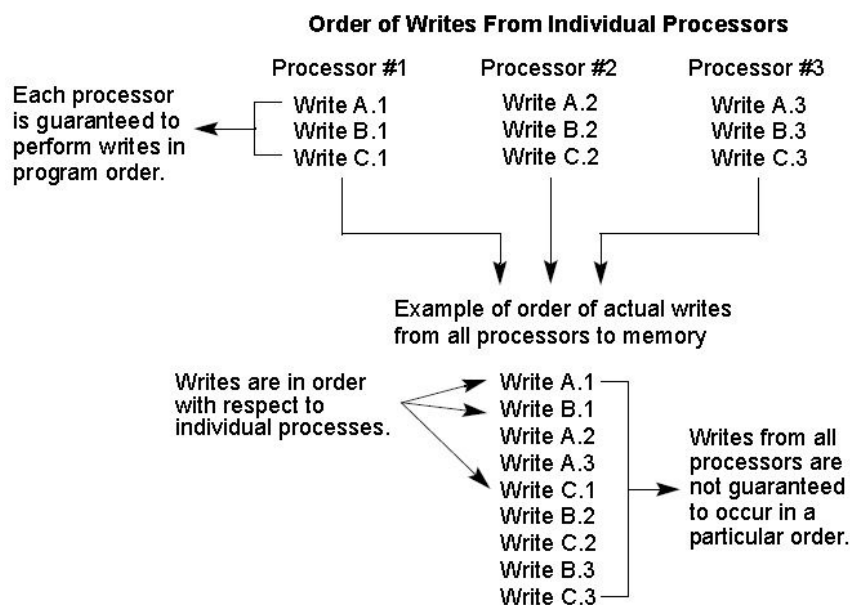


图 6.1 多处理器的内存有序化

6.3 多处理器的初始化

对于 Intel Xeon 处理器来说，多个处理器的初始化工作是驱动中最难实现的部分，在 IA-32 架构中定义了一个 MP（multiprocessor）初始化协议^{[10][32][33][34]}。遵循这个协议我们可以很好完成我们驱动中初始化的工作，MP 初始化协议符合以下特点：

- （1）支持受控的多处理器启动，不需要专用的系统硬件；
- （2）允许硬件直接开始系统启动，不需要专门的信号或者预定义的启动处理器；
- （3）允许所有 IA32 处理器以相同方式启动；
- （4）MP 初始化协议同样适用 Intel64 处理器；

MP 初始化协议必须遵守的两个要求或限制：

（1）MP 初始化协议仅在系统上电或重启时候执行，如果 MP 协议完成了且 BSP 被选中，接下来的初始化操作就不会再引起 MP 协议重复执行。

（2）MP 初始化协议执行的时候所有的设备都不能给处理器发送中断。

MP 初始化协议会根据不同的处理器家族而不同，对于 Intel Xeon 处理器，BSP（bootstrap

processor) 和 AP (application processor) 的选择是通过在系统总线上使用 BIPI (broadcast inter-processor interrupt) 仲裁来实现的。

6.3.1 BSP 和 AP

BSP 和 AP 是两种级别的处理器：引导处理器和应用处理器。系统上电或一个 MP 系统重启之后，系统软件会在系统总线上动态选择一个 BSP，剩下的处理器就会作为 AP。作为 BSP 的处理器 MSR_IA32_APIC_BASE 寄存器的 BSP 标志位会被置 1，在程序代码中我们也可以通过设置此标志位使当前处理器成为 BSP。

BSP 负责执行 BIOS 的 boot-strap 代码来配置 APIC 环境，建立系统数据结构，并初始化 AP。等待 BSP 和 AP 都初始化好之后，然后 BSP 开始执行 OS 初始化代码。系统上电或重启后，AP 会完成最小限度的自身配置，然后等待 BSP 处理器发送一个开始信号，AP 收到信号会执行 BIOS 的 AP 配置程序，接着 AP 会进入停止状态。

在正常执行程序的时候，只有 BSP 是有效的，任何等待执行的程序都由 BSP 处理，但是 BSP 可以调用它所管理的 AP 去执行程序。通过给出 AP 的处理器号码，BSP 就可以启动有效的 AP 去执行等待执行的程序。

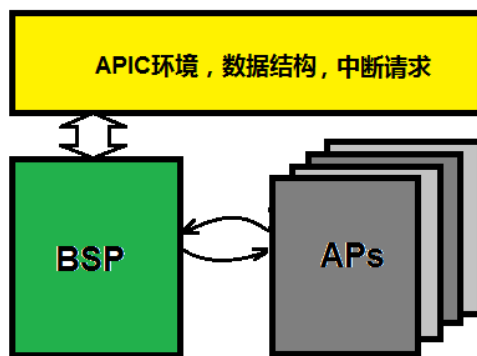


图 6.2 多处理器 BSP 和 AP 模型

6.3.2 Intel Xeon 处理器的 MP 初始化协议算法

系统上电或 MP 系统重启后，处理器的所有处理器都会执行 MP 初始化协议算法来初始化系统总线上的每个逻辑处理器，具体的初始化操作如下^{[10][35]}：

(1) 每个逻辑处理器都被分配一个唯一的 APIC ID，此 ID 是一个写入每个处理器的 local APIC ID 寄存器中的 32-bit 值。每个逻辑处理器都在 APIC ID 中被分配一个唯一的仲裁优先权。

(2) 每个逻辑处理器同步执行它内部的内建自测试 (BIST)。

(3) 一旦 BIST 执行完成, 逻辑处理器给所有包括自己在内的处理器广播一个 BIPI。第一个广播 BIPI 的处理器会选择自己作为 BSP, 并设置 BSP 标志位。其他的处理器的 BSP 标志位被清除, 这些处理器都作为 AP, 然后这些 AP 进入等待 BSP 发送 SIPI (startup IPI) 信号的状态^[36]。

(4) 新确认的 BSP 会去执行 BIOS 的 boot-strap 代码, 创建一个 ACPI 表和一个 MP 表, 并且将 APIC ID 添加到这些表中, boot-strap 程序的最后会给所有 AP 广播 SIPI 信号。

BSP 的确定是由硬件在 SEC 阶段就完成的, BSP 会将自己的 MSR_IA32_APIC_BASE 寄存器的 BSP 标志位置 1, 其他的处理器中该位将被清除。

(5) 所有 AP 竞争获得 BIOS 初始化信号量 SIPI, 获得信号量的 AP 开始执行初始化代码, 将自己的 APIC ID 添加到 ACPI 和 MP 表中, 并将处理器数目加 1。初始化代码执行完后, AP 会进入 halt 状态。

(6) 当每个 AP 都获得了信号量并执行了 AP 初始化代码, BSP 就可以获得系统总线上的处理器数量。BSP 执行完 boot-strap 代码就会开始执行它想要执行的代码。

(7) 当 BSP 执行代码的时候, AP 是处于停止状态的, 只有当 BSP 想要让 AP 执行代码的时候会去唤醒它。

6.3.3 BSP 初始化工作

在 MP 初始化协议算法中已经阐述 BSP 初始化的工作: 创建 ACPI 表和 MP 表, 广播 SIPI 信号给 AP。

ACPI 表是在 SEC 阶段就定义的, 而且是全局的, 它用 EFI_ACPI_TABLE_GUID 来作为入口。关于 ACPI 表的操作如下:

```
struct _EFI_ACPI_TABLE_PROTOCOL {  
    EFI_ACPI_TABLE_INSTALL_ACPI_TABLE        InstallAcpiTable;  
    EFI_ACPI_TABLE_UNINSTALL_ACPI_TABLE      UninstallAcpiTable;  
}
```

BSP 在执行 boot-strap 代码时会执行 _EFI_ACPI_TABLE_PROTOCOL 的接口创建 ACPI 表, 并将自己的 APIC ID 添加进去。(注意 ACPI 是 advanced configuration and power interface, APIC 是 advanced programmable interrupt controller, APIC ID 只是 ACPI 配置信息的一部分)

MP 表同样是在 SEC 阶段创建的, 在 UEFI 中定义为

EFI_LEGACY_MP_TABLE_HEADER, 其中包括处理器的 ACPI ID, BSP 在 boot-strap 中用 GetMpTable()创建和添加 MP 表。

```
EFI_STATUS GetMpTable(....., EFI_LEGACY_MP_TABLE_HEADER *Table, .....
```

BSP 更为重要的作用在于广播 SIPI 信号给处于等待状态的 AP, 然后激活 AP 去执行程序。IPI 是处理器间的中断信号 (inter-processor interrupt), SIPI 是启动处理期间中断信号 (startupIPI), 除了 SIPI 还有 SMIIPI, fixedIPI 和 initIPI。IPI 相当于中断命令, 收到的处理器将作出相应的操作。在 IA-32 的架构中, 每个处理器都会配置一个 APIC, IPI 的发送和接收都是由 APIC 控制的, 在发送信号之前通过设置 APIC 的中断命令寄存器 ICR 可以控制发送的具体操作。

APIC 的 ICR 低 32 比特位控制该处理器产生中断命令 IPI 的向量号, IPI 类型, 目的地等信息。

```
typedef struct {
    UINT32 Vector:8;           //将要发送的中断的向量号
    UINT32 DeliveryMode:3;     //发送 IPI 的类型
    .....
    UINT32 DeliveryStatus:1;
    .....
    UINT32 DestinationShorthand:2; //中断目的处理器的简化符号
    .....
} LOCAL_APIC_ICR_LOW;
```

BSP 在发送 SIPI 需要设置这些比特位, 然后发送 SIPI。

```
void EFIAPI SendInitSipiSipiAllExcludingSelf(
    IN UINT32 StartupRoutine
)
```

具体伪代码如下:

```
{
    LOCAL_APIC_ICR_LOW IcrLow;
    延时 MicroSecondDelay (10);
    //设置将要发送的 IPI 中断命令
    IcrLow.DeliveryMode = LOCAL_APIC_DELIVERY_MODE_STARTUP;
    IcrLow.Bits.Level = 1;
```

```
IcrLow.Bits.DestinationShorthand =除了自己全部发送;  
WriteLocalApicReg (....., IcrLow);  
//通过向本地 APIC 寄存器中写入 IcrLow 启动发送过程  
}
```

6.3.4 AP 初始化工作

AP 在确定之后一直等待 BSP 发送 SIPI 信号，第一个收到信号量的 AP 开始执行初始化代码。它所做的事情和 BSP 类似，也是讲自己的 APIC ID 添加进全局的 ACPI 表和 MP 表中，然后执行一些初始化的汇编指令，最后 AP 将执行一个 CLI 指令是自己处于停止状态，并等待 BSP 再度唤醒它去执行程序。

当所有的 AP 完成初始化之后，BSP 就可以获得系统总线上所有的逻辑处理器的基本信息，包括处理器数目，位置，配置信息。这些信息都在 ACPI 表和 MP 表中。然后所有 AP 都会处于停止状态等待 BSP 去唤醒。

6.4 多处理器的管理

在 Intel 开发多处理器机制之初，有关多处理器的管理一直都是重点要解决的问题。多处理器管理要解决的是：等待执行的程序要交由哪个逻辑处理器去执行？怎样保证各个处理器之间互不干扰的执行程序？其实在多处理器机制中有 BSP 和 AP 两种逻辑处理器，系统中等待执行的程序都交由 BSP 来处理，BSP 可以选择自己执行程序，也可以选择调度其他的 AP 去执行。这就是 BSP 管理 AP 的机制^{[37][38]}。

在 UEFI 的处理器驱动中定义了很多用于多处理器管理的函数 protocol，通过在处理器驱动上安装这些 protocol 接口来实现驱动的功能。

```
EFI_MP_SERVICES_PROTOCOL      mMpService = {  
    GetNumberOfProcessors,  
    GetProcessorInfo,  
    StartupThisAP,  
    SwitchBSP,  
    WhoAmI,  
    ... .. };
```

在 DXE 阶段的处理器驱动主程序中，需要安装这些 MP services protocol 的接口来实现多

处理器管理的功能。启动时服务列表的服务函数 `gBS->InstallMultipleProtocolInterfaces` 用来安装 MP services protocol。

```
Status = gBS->InstallMultipleProtocolInterfaces (
    &驱动生成的 handle,
    &MP services Protocol 的 GUID,
    &mMpService
);
```

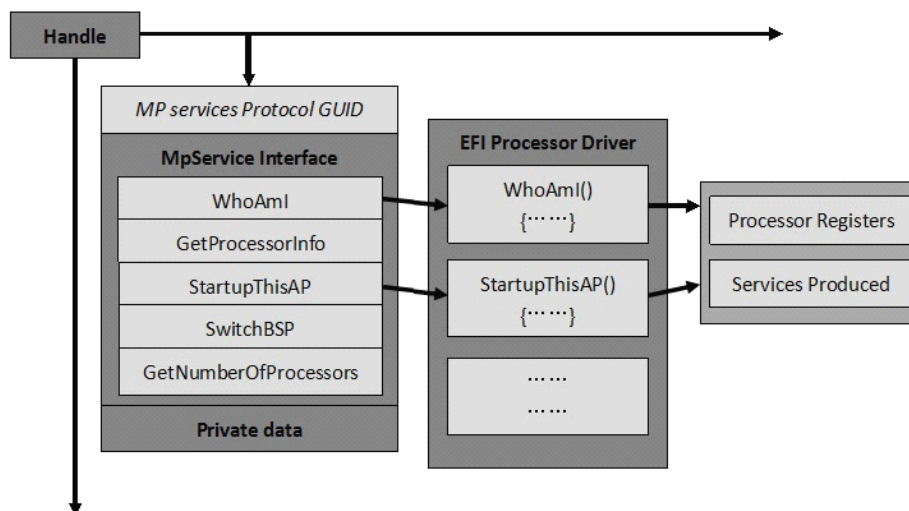


图 6.3 多处理器驱动运行模型

6.4.1 BSP 或 AP 的确定

对于支持 Intel Hyper-Threading 技术的 IA-32 处理器（包括 Intel Xeon）来说，系统总线上的每个逻辑处理器都有一个唯一的 APIC ID。在处理器执行至 DXE 阶段的时候，每个逻辑处理器都会检测自己的 APIC ID，然后和每个逻辑处理器的 APIC ID 逐个对比，找到自己的 APIC ID 之后，就会返回自己的处理器号^[38]。

```
EFI_STATUS  EFIAPI WhoAmI (
    IN      EFI_MP_SERVICES_PROTOCOL  *This,
    OUT     UINTN                      *ProcessorNumber
)
```

具体函数实现的伪代码如下：

```
{
    ApicID=得到对应这个逻辑处理器的 APIC ID;
    for(Index=0; Index<所有检测到的逻辑处理器数量; Index++)
```

```
{
    if(上面得到的 APIC ID 值等于某个 Index 对应的 APIC ID)直接跳出循环;
}
找到它的处理器号并赋给 ProcessorNumber;
返回成功;
}
```

在执行其他服务函数的时候，会首先检测调用者的处理器号是否等于 BSP 号。接下来的所有执行服务函数之前都要去判断该逻辑处理器是否是 BSP 或 AP。

6.4.2 逻辑处理器的信息

在确定 BSP 之后，BSP 会找到所有有效逻辑处理器进行管理。在获得逻辑处理器的信息的时候，定义了一个处理器信息的缓冲区^{[10][39]}。

```
typedef struct {
    UINT 64 ProcessorId;
    UINT32 StatusFlag;
    EFI_CPU_PHYSICAL_LOCATION Location;
} EFI_PROCESSOR_INFORMATION;
```

ProcessorId 是由系统硬件决定的，它相当于 APIC ID，用来唯一标识一个逻辑处理器。它只有低 8bits 有效，高 bit 是预留的。StatusFlag 表示逻辑处理器的状态，只有低 3 位有效，[31:3]是预留的，必须为 0。对于 StatusFlag[2:0]的组合如表 6.1：

表 6.1 statusFlag[2:0]的组合^[10]

BSP	ENABLED	HEALTH	DESCRIPTION
0	0	0	不健康的无效 AP
0	0	1	健康的无效 AP
0	1	0	不健康的有效 AP
0	1	1	健康的有效 AP
1	0	0	无效位
1	0	1	无效位
1	1	0	不健康的有效 BSP
1	1	1	健康的有效 BSP

Location 是结构体，它包括物理的 package 号，即物理处理器所在的插槽，在这个 package 内有几个物理 CPU 核，以及每个核包括几个逻辑线程。

```
typedef struct {
```

```

        UINT32          Package;

        UINT32          Core;

        UINT32          Thread;

    }  EFI_CPU_PHYSICAL_LOCATION;

```

在 MPservice 中的 GetProcessorInfo()用来获得处理器的信息，它通过给定的逻辑处理器号码获得该处理器的上述信息，它只能通过 BSP 调用执行。

```

EFI_STATUS  EFIAPI GetProcessorInfo (
    IN        EFI_MP_SERVICES_PROTOCOL  *This,
    IN        UINTN                      ProcessorNumber,
    OUT       EFI_PROCESSOR_INFORMATION *ProcessorInfoBuffer
)

```

This 是指向 MPservice 的指针，ProcessorNumber 是处理器的号码。函数实现的伪代码如下：

```

{
    (1) 通过 WhoAmI()检查调用该函数的处理器是否为 BSP;
    检查指定的处理器号码是否存在;

    (2) ProcessorInfoBuffer->ProcessorID=获取执行处理器的 APIC ID;
    GetProcessorLocation(指定处理器号, &Location)
    //获取处理器位置信息, 并保存在 Location 中;

    (3) if(处理器的状态是 disabled) {
        ProcessorInfoBuffer->StatusFlag &= ~PROCESSOR_ENABLED_BIT;
        //将指定处理器有效位(第 1 位)清除
    }

    (4) if(处理器号码==BSP 处理器号) {
        ProcessorInfoBuffer->StatusFlag |= PROCESSOR_AS_BSP_BIT;
        //将指定处理器的 BSP 位(第 0 位)置 1
    }

    (5) if(指定处理器是健康的) {
        ProcessorInfoBuffer->StatusFlag |= PROCESSOR_HEALTH_STATUS_BIT;
        //将指定处理器的健康位(第 2 位)置 1
    }
}

```

```
}
```

GetNumberOfProcessors()这个服务函数用来检索平台上所有的逻辑处理器数量，它仅仅会被 BSP 调用。

```
EFI_STATUS  EFIAPI GetNumberOfProcessors (
    IN  EFI_MP_SERVICES_PROTOCOL  *This,
    OUT UINTN                      *NumberOfProcessors,
    OUT UINTN                      *NumberOfEnabledProcessors
)
{
    if(当前为 BSP 在执行)继续; else 直接返回;
    for(Index=0; Index<所有的处理器数量; Index++) {
        得到每个处理器的位置;
        获取每个处理器的 CpuData 数据结构, 这里关键的数据是 CPU 状态;
        if((该处理器支持 Hyper-Threading) &&( 处理器状态不是 disabled))
            (*NumberOfEnabledProcessor)++;
    }
    返回成功;
}
```

对于处理器的运行状态，EFI 为每个运行的处理器定义了一个 CPU_DATA_BLOCK 的结构体，它包括当前处理器正在执行的函数的指针，它正在等待的事件，计时器状态，以及它当前所处的状态。处理器当前的执行状态共有 5 个：空闲，就绪，忙，（切换）完成，以及无效^[40]。

```
typedef struct {
    EFI_AP_PROCEDURE  volatile      Procedure;
    .....
    EFI_EVENT          WaitEvent;
    .....
    SPIN_LOCK          CpuDataLock;
    CPU_STATE  volatile      State;
} CPU_DATA_BLOCK;
```

处理器运行状态对管理多处理器非常重要，尤其在多处理器切换时，要根据处理器当前

状态作出相应的操作。

6.4.3 BSP 和 AP 的切换

自从 Intel 引入了多处理器的机制，BSP 和 AP 的概念就油然而生，BSP 的存在使多处理器之间调度成为可能。但是要注意在 SEC 阶段确定的 BSP 并不是一直作为 BSP 的，BSP 和任何一个 AP 都可以进行角色切换。例如在 BIOS 接下来的阶段会去检查每个处理器的处理性能，如果发现 BSP 的处理性能不如某个 AP，这样就可以将该 AP 提升为 BSP。BSP 和 AP 之间切换的最后结果是 AP 成为新的 BSP，旧的 BSP 成为新的 AP^[29]。

当发生处理器切换时，通常要注意处理器当前的执行状态，需要保持 BSP 和 AP 之间信息传递的完整性，例如 DXE 阶段的计时器数据，本地 APIC 计时器数据以及当前的中断信息，具体的函数声明和算法（伪代码）如下：

```
EFI_STATUS  EFIAPISwitchBSP(
    IN  EFI_MP_SERVICES_PROTOCOL  *This,
    IN  UINTN                      ProcessorNumber,
    IN  BOOLEAN                    EnableOldBSP
)
```

This 是指向 EFI_MP_SERVICES_PROTOCOL 结构体的指针，ProcessorNumber 是要切换的目的处理器号，EnableOldBSP 表示是否要将旧的 BSP 置为无效。

```
{
```

(1) WhoAmI(This, &CallerNumber)检查主调处理器是否为 BSP;

if(CallerNumber 不是 BSP 号)直接返回;

if(ProcessorNumber>处理器个数)目的处理器不存在,返回;

if(ProcessorNumber==BSP 号)切换到自身,返回;

if(处理器状态为 disabled ||busy)返回;

//通过以上检查可以确保没有任何的特殊情况。

(2) 保存当前的 DXE 计时器的数值，以便恢复;

作废 DXE 计数器;

保存 BSP 当前的本地 APIC 计时器的设置;

中止 APIC 计数器中断;

(3) /*因为在切换 BSP 和 AP 角色期间两个处理器共用一个堆栈，一旦发生中断，中断返回地址将被硬件压入堆栈，这将导致堆栈损坏，所以切换期间是不能有中断的*/ (8.3)

```

    OldInterruptState = SaveAndDisableInterrupts ();

    (4) 清除当前 BSP 的 BSP 标志位 (MSR_IA32_APCI_BASE 寄存器中), 它不再是 BSP;
        两个处理器状态=空闲状态;

    (5) 唤醒目的处理器 WakeUpAp ( ProcessorNumber, ... .. );

    (6) 将目的处理器的 BSP 标志位置 1, 它成为新的 BSP;

//由于新的 BSP 的所有寄存器都是旧的 AP 的值, 所以在加载其寄存器时要小心谨慎。

    (7) 将旧的 BSP 的本地 APIC 计时器设置加载至新 BSP;
        恢复之前保存的中断状态到新的 BSP;
        使能和恢复 DXE 计时器的数值;

    (8) //旧的 BSP 此时会在设置一个 SpinLock, 一直等待自己的运行状态改为切换完成。
        while (TRUE) {
            为旧的 BSP 的执行状态上锁 SpinLock;
            CpuState = CpuData->State;
            释放 BSP 执行状态的锁;
            if (CpuState == CpuStateFinished) break;
        }

    (9) ChangeCpuState (BSPNumber(旧的 BSP), EnableOldBSP, ...);
        //根据 EnableOldBSP 决定是否 disable 旧的 BSP

    (10) BspNumber = ProcessorNumber; //更改系统的 BSP 号
        完成 return;
}

```

6.4.4 启动 AP 执行程序

AP 在初始化完成之后, 就会进入停止状态等待 BSP 发送 SIPI 将其唤醒去执行程序。在启动 AP 的过程中特别要注意该 AP 目前所处的运行状态, 必须满足某些条件 BSP 才能让它去执行程序。我们在代码中定义了启动 AP 的函数 StartupThisAP(), 它的具体实现伪代码如下:

```

EFI_STATUS  EFIAPI StartupThisAP (
    IN  EFI_MP_SERVICES_PROTOCOL  *This,
    IN  EFI_AP_PROCEDURE          Procedure,
    IN  UINTN                      ProcessorNumber,

```

```

    IN  EFI_EVENT                      WaitEvent OPTIONAL,
    IN  UINTN                          TimeoutInMicroseconds,
    IN  VOID                           *ProcedureArgument OPTIONAL,
    OUT BOOLEAN                        *Finished OPTIONAL
)

```

This 是目前 BSP 执行过程中产生的 MP 服务 protocol 的指针, ProcessorNumber 是待启动的 AP 处理器号, Procedure 是 AP 将要执行的程序的入口指针, TimeoutInMicroseconds 是启动 AP 处理器的时间限, 其他参数是可选的。

```

{
    (1) CPU_DATA_BLOCK      *CpuData;
        //首先为目标 AP 定义一个区, 记录目标 AP 的运行状态
    (2) //接下来将做相关的检查, 以确保 AP 能顺利唤醒并执行程序
    WhoAmI()保证主调处理器是 BSP;
    if(ProcessorNumber >= 处理器总数目) 目标 AP 不存在;
    if(ProcessorNumber == BSP 处理器号) 目标 AP 为 BSP 自身;
    if(Procedure == NULL) 代执行的程序无效;
    前面定义的 CpuData = &mMPSystemData.CpuData[ProcessorNumber];
    if( CpuData->State == disabled || CpuData->State == Busy) 处理器还没就绪;
    (3) //通过如上检查, AP 可以被启动去执行程序
    为目标 AP 上锁;
    CpuData->State = CpuStateReady;
    解锁;
    CpuData->Procedure = Procedure; //将要执行的程序入口
    SendInitSipiSipiIpi (....., GET_CPU_MISC_DATA (ProcessorNumber, ApicID), .....); //启动 AP 执行程序
    (4) /* 如果 WaitEvent 为空, BSP 将一直检查 AP 状态是否就绪, 直到
    TimeoutInMicroseconds 用完*/
    CpuData->ExpectedTime = TimeoutInMicroseconds+当前时间;
    do {
        Status = 获得目标 AP 的当前状态;
    } while (Status == EFI_NOT_READY);
}

```

```
(5) return;  
}
```

在 BSP 调用 SendInitSipiSipiIpis() 发送 SIPI 之后, 剩下的工作基本都是由硬件来完成, 目的 AP 会接收到 SIPI 并去执行 Procedure, 然后 BSP 会一直等待 AP 执行完毕。

6.5 本章小结

本章主要描述了 Intel IA32 and 64 架构提供的管理和提升系统多处理器的性能的机制, 即多处理器机制。由于本论文所进行的项目所用的 Intel Xeon-E5 处理器也采用多处理器机制, 所以本章是本篇论文的重点。

处理器的多处理器机制需要管理的内容包括: 系统内存的联接, 缓存的一致性, 预先排好写入内存的顺序, 多处理器之间处理中断应用请求, 以及多线程和多进程的开发。本章的主要结构也是按照多处理器机制的内容进行描述的。

上锁的原子操作保证了内存的联接性, 首先系统对内存的读写都是原子性的, 除此之外处理器还提供了对总线上锁的操作, 包括自动上锁和软件控制上锁, 这对关键内存的读写非常有用。内存有序化阐述了多处理器如何实现对内存的访问的有序化, 8.2 节中给出了内存读写有序化的模型。对于对于多处理器的驱动来说, 实现多处理器的初始化以及并行执行程序是最关键的部分, 本章最后两节中重点研究了这两个方面的机制。

对于 Intel Xeon 处理器来说, 在 IA-32 架构的多处理器规范说明书中专门定义了多处理器的初始化机制, 即 MP 初始化协议。在系统有两种处理器: 引导处理器 BSP 和应用处理 AP, BSP 会负责执行 BIOS 的 boot-strap 代码并初始化 AP, 而 AP 只会完成最小限度的自身配置, 然后进入停止状态。针对这两种处理器文中分别描述了各自的初始化算法, 并给出了伪代码上的实现。

系统总线上多处理器的管理是多处理器固件驱动中核心的内容, 也是本论文的最重要的点。多处理器管理实现了多处理器之间的调度和通信机制, BSP 作为系统中的引导处理器, 主要由它来执行这些机制的具体操作。首先系统要确定多个逻辑处理器的角色, 然后通过选中的 BSP 可以获得系统所有逻辑处理器的信息, 也可以实现 BSP 和 AP 的切换, 以及最重要是调度目标的 AP 去执行指定的程序。在本章最后一节的内容中主要是从伪代码的实现上描述了这些程序实现的机制。

第七章 总结与展望

7.1 课题总结

本文基于目前最新的 EDKII/Tiano 体系架构, 针对 Intel IA-32 and 64 体系架构 Xeon-E5 多处理器提出了 UEFI 固件驱动的设计方法, 并给出了具体的程序实例。主要工作具体有以下几个方面:

(1) 研究了 UEFI 和 EDKII/Tiano 技术, 阐述了其技术背景、体系结构、运行机制、主要功能以及存在的问题等。

(2) 根据 UEFI 定义的标准规范以及 Intel IA32 and 64 体系架构的定义, 基于最新的 EDKII/Tiano 的框架, 设计平台上 Xeon-E5 多处理器固件驱动的开发, 并给出具体的开发工具和开发流程。

(3) 通过对 IA32 and 64 体系架构的内存管理机制的研究, 本文分别分析了分段和分页机制。通过设计程序为特定任务准备其执行环境中各个段的段选择子和段描述符, 这样就可以将多任务的执行环境隔离开, 实现了分段功能; 通过设计程序为地址转换过程准备分页结构体入口, 使任务执行环境的线性地址转换成物理地址, 实现分页功能。

(4) 研究了处理器中断和异常管理机制, 通过将结构化定义的中断或异常情况与中断向量号相对应, 然后通过执行中断或异常处理程序实现中断或异常情况的响应。文中重点研究了中断和异常响应的过程, 并通过具体实例介绍了如何管理一个中断或异常情况。

(5) 分析了 IA32 and 64 体系架构的任务管理机制, 研究了任务管理机制的如下功能: 保存一个任务的状态信息, 调度任务执行, 和切换不同的任务。文中重点研究了任务管理功能的任务切换机制, 并设计程序实现了任务切换功能。

(6) 针对多处理器管理机制, 研究了多处理器管理机制所实现的功能。对于本课题的多处理器驱动程序的设计, 重点研究了多处理器的管理机制和多处理器之间的调度机制, 并着重用程序描述了这些机制实现的过程。

论文中从第三章至第六章分别研究了多处理器固件驱动的内存管理、中断和异常管理、多任务管理和多处理器管理四个主要模块, 根据对各个模块的实现机制的分析, 可以看出这四者是有关联的。总结其关联性如下:

(1) 内存管理是处理器对内存资源的管理, 是最基本的功能, 任何想要访问内存资源的操作都会使用内存管理。如在多任务管理机制中保存和加载一个任务的状态信息就需要访问

和读写内存空间。尤其对于任务切换时，处理器要保存旧的任务状态段 TSS，然后加载新的任务状态段 TSS，要遵循从段选择子到段描述符再到 TSS 的分段过程。

(2) 中断或异常处理机制和多任务管理机制是相互包含的关系。在 4.9.2 中已经描述了当中断向量号索引到一个任务通道描述符时，处理器会去执行中断服务程序，这实际上也是多任务之间的切换过程；另一方面在任务切换机制中隐含调用中断或异常服务程序就是中断处理程序的响应过程 (5.2)。

(3) 对于多处理器管理机制，其原理是在宏观上管理每个逻辑处理器，每个逻辑处理器都需要具备以上三者的功能机制。

7.2 研究展望

本文的内容紧紧围绕着 Intel Xeon-E5 多处理器 UEFI 固件驱动的各个模块的完成而进行，这里面还有很多工作可以继续展开：

(1) 针对 UEFI 标准的 EDKII/Tiano 框架，还可以进一步研究框架组成部分之间的关系，可以重点研究在编译生成 bin 可执行文件时各组件的连接关系。

(2) 对于处理器的内存管理机制，可以研究其他的分页模式，例如 PAE 模式，IA-32e 模式。尤其对于 IA-32e 分页模式来说，其分页层级可以更加细分，映射到物理地址的分页大小也可以为 4-KByte、2-MByte 和 1-GByte，可以分情况研究具体的实现机制以及程序实现。

(3) 对于外部中断的响应过程，还可以进一步研究 I/O APIC 和本地 APIC 管理外部中断的机制。对于 IA32 and 64 体系架构来说，高级可编程中断控制器 APIC 不仅用于管理所有外部中断的触发，同时它还和逻辑处理器通信并传送外部中断信号，这是该体系架构重要的一部分。

(4) IA32 and 64 体系架构的电源管理也是比较重要的模块，随着处理器性能的不断升级，功耗的降低也是至关重要的，这部分内容主要由高级配置和电源接口 ACPI 控制。

参考文献

- [1] 洪蕾. UEFI 的颠覆之旅[J]. 中国计算机报, 2007(50): 1-3.
- [2] 邢卓媛. 基于 UEFI 的网络协议栈的研究与改进[D]. 上海:华东师范大学, 2011.
- [3] Vincent Zimmer. Beyond BIOS[M]. The United States: Intel Corporation, 2006:17-32, 143-146.
- [4] 倪光南. UEFI BIOS 是软件业的蓝海[C]. UEFI 技术大会, 2007: 1-5.
- [5] 刘江. 基于 UEFI 的硬件信息检测和性能分析技术的研究与实现[D]. 苏州:中国科学技术大学, 2009.
- [6] Gaurav B. EFI/UEFI 将带领 PC 产业进入下一代 [J/OL]. http://www.22cc.net/news/20080506/17634_2.html, 2008-05-04.
- [7] 崔莹, 辛晓晨, 沈钢钢. 基于 UEFI 的嵌入式驱动程序的开发研究[J]. 计算机工程与设计. 2010, 31(10):2384-2387.
- [8] Intel Corporation. EDKII user manual[M]. Revision 0.5. The United States: Intel Corporation, 2008: 10-12.
- [9] Framework Open Source Community. Edk Getting Started Guide[M/OL]. Revision 0.41. <https://efi-shell.tianocore.org/servlets/ProjectDocumentList>, 2005:11-15.
- [10] Intel Corporation. Intel 64 and IA-32 Architectures Software Develop's Manual[M]. The United States: Intel Corporation, 2009:10-778.
- [11] Baraz L, Devor T, Etzion O. IA-32 Execution Layer: a two-phase dynamic translator designed to support IA-32 applications on Itanium®-based systems[C]. Proceedings of the 36th International Symposium on Microarchitecture (MICRO-36 2003), 2003:1102-1113.
- [12] 湛辉来, 曾一. X86 体系中保护模式下的内存访问机制[J]. 重庆大学学报(自然科学版), 2002, 25(6):67-70.
- [13] Intel Corporation. EFI Debugging using ITP[M]. Revision 0.3. The United States: Intel Corporation, 2006:15-17.
- [14] UEFI Forum. UEFI Specification[M]. Version 2.3.1. The United States: Intel Press, 2009:5-56.
- [15] 赵丽坤, 于德海, 王阳. Linux 内核内存管理机制和改进[J]. 电脑知识与技术, 2008, 4(3):752-753.
- [16] 封斌, 龚灼. 实时操作系统保护模式下的内存管理策略[J]. 华中科技大学学报(自然科学版), 2002, 30(3):94-96.
- [17] Gnadeberg J B, Zeckendorf L J. Paging in hierarchical memory systems[P]. The United States, 3984818.1976-10-5.
- [18] John H et al. Method and apparatus for a hierarchical paging storage system[P]. The United States, 4430701.1984-2-7.
- [19] Song et al. System and method for handling interrupt and exception events in an asymmetric multiprocessor architecture[P]. The United States, 6003129.1999-12-14.
- [20] Wing-Chi P, Aloysius K. Bounding the Running Time of Interrupt and Exception Forwarding in Recursive Virtualization for the x86 Architecture[C]. Technical Report VMware-TR-2010-003, 2010:103-116.
- [21] 沈雪峰. 多 CPU 系统的中断机制[D]. 成都: 电子科技大学, 2009.
- [22] Buhr P et al. Advanced exception handling mechanisms[C]. IEEE Transactions On Software Engineering, 2000:820-836.
- [23] James M. A language to enable supervisor-level microthreading on the x86[D]. The United States: The University of York, 2010:69-78.
- [24] Pablo R. Efficient Implementation of the bare-metal Hypervisor MetalSVM for the SCC[C]. The 6th Many-core Applications Research Community (MARC) Symposium, 2012:59-65.
- [25] Rajeev D et al. METHOD, APPARATUS AND SYSTEM TO SAVE PROCESSOR STATE FOR EFFICIENT TRANSITION BETWEEN PROCESSOR POWER STATES[P]. The United States, 13/075,057.2011-3-29.

- [26]高小明. 基于 Intel VT 硬件虚拟机内核研究与实现[D]. 成都: 电子科技大学, 2010.
- [27]Irvine K R. Assembly language for x86 processors[M]. Sixth Edition. Florida:Florida International University, 2009:20-21.
- [28]Glenn H etal. MICROPROCESSOR THAT FACILITATES TASK SWITCHING BETWEEN ENCRYPTED AND UNENCRYPTED PROGRAMS[P]. The United States,13/091,698.2011-4-21.
- [29]Intel Corporation. Multiprocessor Specification Version 1.4[M]. The United States: Intel Corporation, 2009:10-778.
- [30]Colin B, Martin M M,Thomas F W. Performace-Transparent Memory Ordering in Conventional Multiprocessors[C]. ISCA'09,2009:1-11.
- [31]Jeffrey A etal. Global Stores and Atomic Operations[P]. The United States, 12/849,766.2010-8-3.
- [32]宋秀兰, 吴晓波. 多处理器通信机制设计[J]. 浙江工业大学学报, 2010, 38(4): 426-429.
- [33]黄安文, 高军, 张民选. 多核处理器非一致 Cache 体系结构延迟优化技术研究综述[J]. 计算机研究与发展, 2012,49(z1):118-124.
- [34]郭超, 李坤, 王永炎等. 多核处理器环境下内存数据库索引性能分析[J]. 计算机学报, 2010, 33(8): 1512-1522.
- [35]Robert I, Alan B. A survey of hard real-time scheduling for multiprocessor systems[J]. ACM Computing Surveys, 2011, 43(4):1743-1749.
- [36]Susmit S, Peter S, Tom R etal. The semantics of X86-CC multiprocessor machine code[C]. Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages, 2009: 379-391.
- [37]陈冠诚. 多线程编程中的原子操作[J]. 程序员, 2012(3): 116-119.
- [38]Levin G. DP-FAIR: A Simple Model for Understanding Optimal Multiprocessor Scheduling[C]. 2010 22nd Euromicro Conference on Real-Time Systems (ECRTS),2010:3-13.
- [39]陈友贵, 王兆平. Linux 同步机制研究[J]. 电脑知识与技术(学术交流), 2010(2): 886-888.
- [40]Pablo M, Matthew H, Samuel T etal.Capo:a software-hardware interface for practical deterministic multiprocessor replay[C]. Proceedings of the 14th international conference on Architectural support for programming languages and operating systems,2009:73-84.

附录 1 攻读硕士学位期间参加的科研项目

- (1) 江苏苏源光一科技有限公司委托的横向课题，负载控制管理中断平台改造（KH0020311029）。

致谢

本课题在选题及研究过程中得到了校内导师邱晓晖教授和企业导师卫治国工程师的悉心指导。本论文从选题、实现、到撰写，邱老师为我指点迷津，帮助我开拓研究思路，她的严谨求实的态度，踏踏实实的精神，深深地感染了我。

感谢 Intel 亚太研发有限公司的企业导师卫治国，在论文项目和实习项目中，卫工都给予了大量的技术支持。同时感谢实习项目组的吴鹏、钱春阳、陶恒燕、周小虎，他们也给予了大力的支持。没有他们在专业培训、研究设备以及工作时间上提供的支持和帮助，该论文将无法完成。

感谢通信工程学院的各位老师对我的教育培养，感谢他们细心指导我的学习与研究。感谢校内实验室的同门朋友：杜娟，吴贺猛，聂和平，吴跃明，连涛，谢谢你们在学校学习期间给我的帮助，以及企业实验室的朋友：庞敏，钱春阳，戴领，郝栋，傅雷，赵欣培，谢谢你们在我实习期间给我的帮助和支持，我们一起度过了一段快乐的时光！

最后，向我的父亲、母亲致谢，感谢他们这些年来对我的辛苦付出。