申请上海交通大学硕士学位论文

基于多核处理器架构的嵌入式微内核操作系统的研究与设计

专　　业：电路与系统

研究方向：嵌入式系统

姓　　名：张荫芾

导　　师：周玲玲　副教授

上海交通大学电子信息与电气工程学院

二〇〇九年一月

A Thesis Submitted to Shanghai Jiao Tong University

For The Degree of Master of Engineering

RESEARCH AND DESIGN OF EMBEDDED MICRO-KERNEL OPERATING SYSTEM BASED ON MULTI-CORE ARCHITECTURE

Major: Circuit and Systems

Field of Interest: Embedded System

Name: Zhang Yinfei

Advisor: Zhou Lingling

School of Electronic, Information and Electrical Engineering

Shanghai JiaoTong University

January, 2009

上海交通大学
学位论文原创性声明

本人郑重声明：所呈交的学位论文，是本人在导师的指导下，独立进行研究工作所取得的成果。除文中已经注明引用的内容外，本论文不包含任何其他个人或集体已经发表或撰写过的作品成果。对本文的研究做出重要贡献的个人和集体，均已在文中以明确方式标明。本人完全意识到本声明的法律结果由本人承担。

学位论文作者签名：

日期：　　年　　月　　日

上海交通大学
学位论文版权使用授权书

本学位论文作者完全了解学校有关保留、使用学位论文的规定，同意学校保留并向国家有关部门或机构送交论文的复印件和电子版，允许论文被查阅和借阅。本人授权上海交通大学可以将本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和汇编本学位论文。

保密□，在___年解密后适用本授权书。

本学位论文属于

不保密□。

（请在以上方框内打“√”）

学位论文作者签名：　　指导教师签名：

日期：　　年　　月　　日　　日期：　　年　　月　　日

基于多核处理器架构的嵌入式微内核操作系统的研究与设计

摘 要

随着嵌入式行业的飞速发展，嵌入式设备中的操作系统的性能也越来越受到重视。可靠性、实时性及安全性是嵌入式系统最需要的性能，但传统的单内核操作系统在这些方面远不如微内核操作系统有优势。以 L4 为代表的第二代微内核操作系统在解决了 IPC 的性能问题后，很好地适应了嵌入式设备的需求，但微内核操作系统目前尚不支持多核处理器架构，然而，现实却是多核处理器架构在嵌入式领域的普及已经成为一种趋势。本文的出发点正是在 L4 微内核操作系统的基础上，采用改进和扩展的方式，设计一个适用于多核嵌入式平台的微内核操作系统，以促进多核处理器架构在嵌入式领域的深入应用。

针对嵌入式系统的特点，本文提出将分离模型的概念应用到整体设计，也即在每个处理器内核上运行一个操作系统内核，多个操作系统内核之间通过一定机制同步。在内存管理方面，本设计允许同一个任务的多个线程同时运行在多个处理器内核上，也即地址空间的覆盖域可以跨核，但对于保存地址空间映射关系的 MapDB，只有一个实例并被所有的操作系统内核共享。在线程调度机制上，本设计允许处理器核心内部的细粒度调度，并提供线程迁移的机制来实现多核间的调度。在线程间通信方面，本文设计了一个邮箱系统作为多核间的 IPC 机制，并通过代理线程的方式实现透明的调用。对于微内核之上运行的服务应用程序，本设计提供独立模式、主从模式和分布式模式三种选择。

本文完成的基于多核处理器架构的微内核操作系统设计非常适用于嵌入式环境。设计中使用的邮箱系统充分利用了多核处理器架构共享 L2 Cache 的特点，性能分析证明本设计能提高跨核 IPC 的速度，进而提高整个系统的性能。将分离模型应用到多核系统的整体设计思路，使得本设计的操作系统在嵌入式平台的可靠性、实时性和安全性方面有很多应用场景。本文的设计不仅应用于与 Intel 公司合作的基于微内核的虚拟化技术研究项目中（Intel 大学合作项目），也为其它微内核操作系统设计提供了参考。

关键词： 操作系统，微内核，多核处理器，嵌入式系统

RESEARCH AND DESIGN OF EMBEDDED MICRO-KERNEL OPERATING SYSTEM BASED ON MULTI-CORE ARCHITECTURE

Abstract

With the fast development of embedded system industry, the performance of the operating systems on the embedded devices is more and more important. Reliability, real-time and security are the three most wanted feature of embedded operating system and on these three aspects, micro-kernel operating systems have a better performance comparing to traditional monolithic-kernel operating systems. L4, as the representative of second generation micro-kernel operating system, not only greatly improves its IPC efficiency, but also fits the requirements of embedded systems very well. However, current micro-kernel operating system cannot support multi-core architecture, while multi-core architecture is becoming more and more popular in the embedded system field. The task of this paper is to design a micro-kernel operating system which is suitable for multi-core embedded platform based on the current L4 micro-kernel.

Based on the characters of the embedded systems, this paper adopts the concept of split model, which deploys a micro-kernel on each processor core, as the general design. There is also synchronization between these kernels to make them work together. For memory management, the design of this paper allows different threads of the same task running on different processor cores, which means that the address space of the task can be across the kernel boundary. On the other hand, the structure used to save address space mapping relationship, MapDB, will have only one instance and shared by all kernels. For thread schedule, the design of this paper limits the fine-grain schedule inside the processor core and provides a thread migration mechanism for cross-core schedule. For inter thread communication, this paper designs a mailbox system for cross-core communication and transparency is realized

through proxy threads. For service applications running on micro-kernel, the design of this paper provides independent mode, master-slave mode and distributed mode.

This paper provides a design of micro-kernel operating system based on multi-core architecture, which fits embedded environment very well. The mailbox system design fully leverages the shared L2 Cache character of multi-core architecture and has a good performance on cross-core communication. The idea of using split model in the general design provides the operating system a lot of application scenarios that fit embedded system's reliability, real-time and security requirements. The design in this paper is not only used in the project cooperated with Intel: Enhanced Virtualization Based on Micro Kernel Architecture (Intel university sponsored project), but also provides a reference design for other micro-kernel operating systems.

Keywords: Operating System, Micro-kernel, Multi-core, Embedded System

目 录

第 1 章 绪论 1

1.1 背景介绍 1

1.2 现有研究成果 3

1.3 设计目标 4

1.4 研究内容 6

1.5 软硬件平台 7

1.6 论文结构 7

第 2 章 多核架构和微内核操作系统介绍 9

2.1 多处理器架构和多核架构的比较 9

2.2 微内核操作系统 10

2.2.1 微内核操作系统架构 10

2.2.2 微内核操作系统优点 11

2.3 L4 规范 13

2.4 FIASCO 14

2.5 本章小结 16

第 3 章 总体架构设计 17

3.1 L4 系统架构 17

3.1.1 L4 内核功能 17

3.1.2 L4 的系统调用 18

3.2 全局模型 20

3.3 分离模型 21

3.3.1 离散模型架构 21

3.3.2 分离模型应用场景 25

3.4 本章小结 25

第 4 章 线程间通信 26

4.1 L4 的 IPC 机制 26
4.1.1 IPC 分类 26
4.1.2 IPC 流程 28
4.2 多核间 IPC 30
4.3 邮箱系统 31
4.4 本章小结 34
第 5 章 系统功能设计 36
5.1 内存管理 36
5.1.1 L4 的内存管理 36
5.1.2 跨核地址空间共享 38
5.2 线程调度 39
5.2.1 L4 调度策略 39
5.2.2 核间调度 40
5.3 锁和同步机制 42
5.3.1 L4 的同步机制 42
5.3.2 多核间同步 43
5.4 服务应用程序 44
5.4.1 独立模式 44
5.4.2 主从模式 45
5.4.3 分布式模式 46
5.5 本章小结 48
第 6 章 系统功能实现 49
6.1 环境、设备及方法 49
6.2 邮箱系统功能实现 49
6.3 邮箱系统性能分析 52
6.4 系统性能测试 53
6.5 本章小结 56
第 7 章 总结与展望 57

7.1 主要结论 57
7.2 研究展望 58
参考文献 60
致 谢 62
攻读学位期间发表的学术论文 63

缩略语表

ABI	Application Binary Interface	应用二进制接口
APIC	Advanced Programmable Interrupt Controller	高级可编程中断控制器
IPC	Inter-Process Communication	进程间通信
IPI	Inter Processor Interrupt	处理器间中断
MESI	Modified, Exclusive, Shared, Invalid (Protocol)	共享、独占、共享、无效（协议）
MID	Mobile Internet Device	手持互联网设备
OS	Operating System	操作系统
PPE	Power processing element	主处理单元
SMP	Symmetric Multi Processor	对称多处理器架构
SPE	Synergistic Processing Elements	协处理单元

插图索引

图 1-1 嵌入式平台操作系统占有率比较图（来源：www.linuxdevices.com） 2
图 2-1 多处理器架构（左）和多核架构（右）的比较 9
图 2-2 单内核操作系统（左）和微内核操作系统结构比较图 10
图 2-3 操作系统中的错误在各模块中的分布（来源：文献[8]） 12
图 2-4 Linux 和 L4Linux 性能比较（来源：文献[11]） 13
图 2-5 DROPS 操作系统层次图（来源：文献[14]） 15
图 3-1 L4 系统架构层次图 17
图 3-2 全局模型示意图 21
图 3-3 离散模型结构示意图 22
图 3-4 离散模型层次示意图 23
图 4-1 IPC 相关概念的继承关系图 27
图 4-2 IPC 发送流程图 28
图 4-3 IPC 接收流程图 29
图 4-4 邮箱系统示意图 32
图 4-5 引入代理线程后的邮箱系统示意图 33
图 5-1 地址空间授予和映射操作示意图 37
图 5-2 fpage 字各位含义示意图 37
图 5-3 跨核调度机制示意图 41
图 5-4 独立模式示意图 45
图 5-5 主从模式示意图 46
图 5-6 分布式模式示意图 47
图 6-1 系统调用类图 50
图 6-2 IPC 系统调寄存器含义示意图 50
图 6-3 线程号说明图 51
图 6-4 基于寄存器 IPC 的执行时间分布图 55

表格索引

表 2-1 L4 各实现版本比较（来源：文献[13]） 14
表 6-1 缓存访问速度比较..... 53
表 6-2 关键 x86 汇编指令执行时间表..... 54
表 6-3 IPC 执行时间表 55

第1章 绪论

1.1 背景介绍

所谓嵌入式系统是一种完全嵌入受控器件内部，为特定应用而设计的专用电脑系统，通常执行的是带有特定要求的预先定义的任务。最近几年，嵌入式技术得到了飞速的发展。从汽车到工业设备，从手机、mp4 等消费电子产品，到电冰箱、洗衣机等传统电器，到处可以看到嵌入式设备的身影。根据 BCC Research Group 的分析结果，嵌入式行业在最近 5 年中保持着年均 14%的增长，到 2009 年将会成为一个 880 亿的巨大市场。

嵌入式平台上运行的操作系统一方面会根据嵌入式平台所执行的功能对内核组件进行裁剪，在满足系统的应用功能的基础上去除不必要的部分；另一方面，嵌入式平台对操作系统的可靠性、实时性和安全性等特性有很高的要求。

可靠性是指嵌入式设备特别是工业设备中的嵌入式系统经常需要连续运行数以年计的时间而不出差错。可以想象，如果飞机中的嵌入式系统在飞机飞行时崩溃重启，会造成多么大的危害。这就要求嵌入式系统上的操作系统的运行完全没有错误，或者在错误出现的时候可以快速自动复位，并且避免在操作系统中使用不稳定的模块。

实时性是指系统能在确定的时间内执行操作并对外部的异步事件做出响应，比如汽车发生车祸时安全气囊必须在极短的时间内打开。一次正确的操作不仅要求逻辑功能上的正确，而且要求完成这些操作所花费的时间在限定之内。实时又分为硬实时和软实时，硬实时要求任务在规定时间内必须完成，这由操作系统来保证；而软实时要求事件响应是实时的，并按照任务的优先级，尽可能在短时间内完成任务。实时操作系统首先需要调度一切可利用的资源完成有实时性要求的任务，其次才考虑提高操作系统的整体效率。

随着嵌入式系统越来越多地与外部连接，甚至是通过互联网连接，其安全性也越来越受到关注。比如用掌上电脑进行网上购物的时候，用户的银行帐号信息必须得到严格的保护。安全性具体是指要求嵌入式设备在与外部连接的过程中，

其内部的数据不会偶然或被恶意地破坏、更改或者泄露，维持嵌入式系统中信息的保密性和完整性。

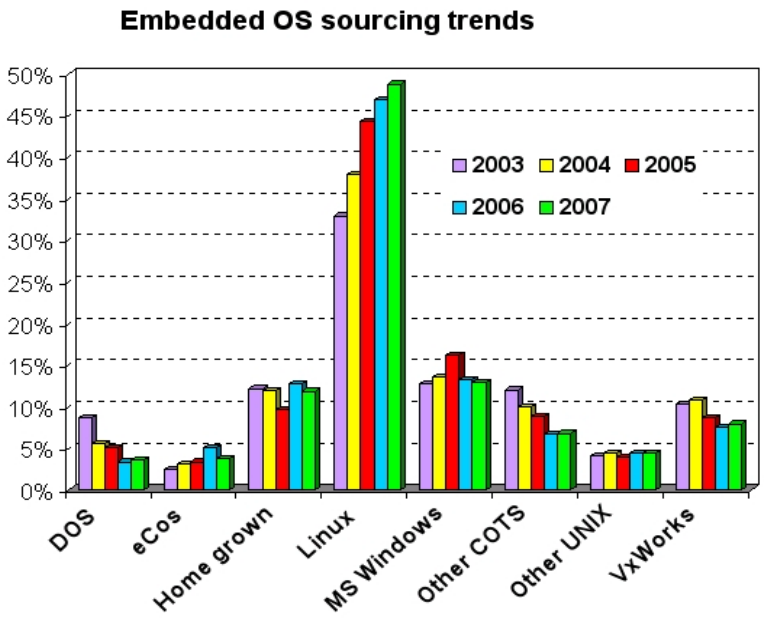

图 1-1 嵌入式平台操作系统占有率比较图（来源：www.linuxdevices.com）

Fig. 1-1 Embedded system OS market occupation chart (Source: www.linuxdevices.com)

说到嵌入式系统上的操作系统，人们首先想到的是各种经过裁剪的 Linux 以及 WinCE，如图 1-1 所示，Linux 也是在嵌入式领域被用得最多的操作系统。但是 Linux 和 WinCE 本身作为单内核操作系统（monolithic kernel），内核部分过于复杂，并不适用于嵌入式平台。而 μC/Os-II 和 eCos 等又过于简单，不能提供完整的功能。而随着微内核（microkernel）操作系统的理论和实现越来越成熟，微内核操作系统开始成为嵌入式平台一个很好的选择。所谓微内核操作系统是指在内核中只提供必要的服务，而把其它服务全部放到用户态的操作系统。其中必要的服务包括进程/线程的调度，进程间通信以及内存的映射。而对于驱动程序、文件系统、内存分配等则放到了用户态的服务程序来执行。

虽然微内核操作系统非常适用于嵌入式环境，但大多数微内核操作系统尚不支持多核的硬件架构。多核架构是指在一枚处理器中集成两个或多个完整的计算单元(内核)。随着计算机技术的飞速发展，多核处理器已经不再是服务器、工作站级别计算机所特有的奢侈品，而是普通个人电脑的配备。可以预见不久的将来，多核处理器也将在嵌入式领域普及。事实上，Intel 最新推出的适用于掌上电脑

的凌动系列芯片已经有了双核的版本。

本论文研究的出发点正为嵌入式设备设计支持多核架构的微内核操作系统。

1.2 现有研究成果

L4 作为第二代微内核操作系统的代表，在工程界已经有了一定的应用，而在学术界也是比较热门的研究对象。从已查阅的文献资料看，不少学者对于将 L4 微内核操作系统移植到多处理器架构上作了研究，但由于 Intel 多核处理器架构还是相对较新的技术，尚无发现有学者针对多核处理器架构上的微内核操作系统进行研究。本论文所做的工作，正是针对多核处理器架构，设计嵌入式环境下的微内核操作系统。其它学者在多处理器架构上的研究，对本论文的帮助在于，一方面学习他们对于微内核架构的分析及各种设计的长处和短处；另一方面借鉴一些有创意的设计思想，应用到基于多核架构的微内核操作系统设计上。因此分析多处理器架构微内核操作系统的相关文献对本文的研究是非常有益处的。

Karlsruhe 科技大学的 Marcus Völp 设计了基于 SMP（Symmetric Multi Processor）架构的 L4 微内核操作系统[1]。他提出使用一个多处理器兼容的用户态的调度器，这个调度器可以把某个线程从某个处理器转移到另一个指定的处理器上执行。此外他还提出使用 L4 规范中原本的 IPC 机制的一个未被使用的标志位来标识某个 IPC 操作是仅在处理器内部操作还是需要传递给在另一个处理器上的线程。

Chemnitz 科技大学的 Sven Schneider 提出了分离模型的概念[2]。分离模型把微内核操作系统的内核复制多份，而在每个处理器上都运行一个内核的实例。本论文的设计也借鉴了这个思想，但本论文的分离模型和 Sven Schneider 提出的有较大的不同。Sven Schneider 的分离模型更多的是从减少工作量的角度出发，每个处理器上的操作系统内核相对独立，不允许线程从一个处理器迁移到另一个处理器，也不允许同一个任务的多个线程运行在多个处理器上，也即不可能存在并行计算。而本文的分离模型主要从嵌入式环境的应用角度出发，并充分利用多核架构共享 L2 Cache 的特性，多个处理内核上的操作系统之间有高效的同步和通信机制。

Karlsruhe 科技大学的 Volkmar Uhlig 在他的博士论文中分析了微内核构操作系统的各种扩展方向[3]，其中就包括了在多处理器架构上的扩展。针对多处理器上的线程间同步的需要，他提出了称之为自适应锁的机制，这种自适应锁可以在运行时动态地自行决定锁的范围。自适应锁可以有两种作用域：处理器本地作用域和跨处理器作用域。应用程序在运行过程中会告知内核应该使用哪个作用域以及哪个处理器有权限访问自适应锁。此外他还在论文中讨论了 L4 的地址空间映射数据库 MapDB 的共享方式。他提出用一个特别的锁来处理地址空间映射树的同步，这个锁的作用域可以仅是映射树的一部分。这样就允许不同处理器上的多个线程同时访问地址空间映射树的不同部分，提高了效率。

综上所述，上面的这些研究虽然是针对多处理器架构上的 L4 微内核操作系统设计，但一些设计思想比如分离模型还是可以应用到多核架构的微内核操作系统设计上。而文献中一些对各种设计方案的取舍分析也对本设计有很大的启发。

1.3 设计目标

本文的研究内容是设计一个支持多核处理器架构的微内核操作系统，除了功能上满足要求之外，本设计还以如下的设计目标作为约束准则。

高性能

微内核操作系统最大的缺陷在于性能，而以 L4 为代表的第二代微内核操作系统最大的改进也在大大提高了系统的性能，特别是 IPC 部分的性能。本文在对基于多内核处理器的微内核操作系统进行设计时，首先考虑的因素也是性能。针对多核处理器架构共享 L2 Cache 的特点，本文特意设计了邮箱系统来提高跨核 IPC 的性能。

兼容性

L4 微内核操作系统上已经开发了许多应用程序，本设计的目标是将这些应用程序应用到新的支持多核架构的微内核操作系统中，并且依旧运行在单核处理器的架构上，或者虽然运行在多核处理器架构上，但作用域仅限于某个处理器内

核内部的场景时，应用程序不需要特别的修改而依旧可以使用。

透明性

所谓透明性是指基于多核的微内核操作系统应该为底层的系统调用提供统一的接口，而使用系统调用的应用程序本身不需要特别区分使用多核版本的系统调用还是处理器内核本地版本的系统调用，而是由操作系统内部来加以判断和选择。

无退化

在增加了对多核处理器架构的支持之后，对于那些仅仅在处理器内核本地进行的操作，性能只能容忍轻微的下降。也即操作系统在单内核处理架构的场景下性能无明显退化，毕竟在实际应用中，处理器内核内部的操作数量要大大超过跨处理器内核的操作。

实时性

微内核操作系统的一大特性是提供完备的实时性支持，而 L4 规范的诸多实现版本中，Fiasco 又以在实时性方面的性能著称。因此在为 Fiasco 微内核操作系统增加了对多核架构的支持之后应该依旧保持其实时特性，保证有限时间长的中断响应和任务执行时间，并且保证实时调度模型也能应用到多核架构下。

复杂性

微内核操作系统最大的特点就是内核的体积非常小。本设计为微内核操作系统增加对多核处理器架构的支持不可避免地会增加操作系统内核的代码量。本设计秉承微内核操作系统的设计理念，也就是把操作系统的相关服务作为用户态的应用程序，而在内核内部只提供实现这些服务的基本机制，并且在兼顾功能和性能的基础上尽量降低操作系统各种机制的复杂程度。

1.4 研究内容

如前文所述，微内核操作系统因为其在可靠性、实时性和安全性方面的优势，在嵌入式环境中有很大的应用；然而随着多核架构开始在嵌入式平台上流行，嵌入式设备需要操作系统能对多核处理器架构进行支持，但现在尚无微内核操作系统支持多核架构。本文的研究内容正是设计一个能支持多核处理器架构并且适用于嵌入式环境的微内核操作系统。

经过调研发现，L4 作为第二代微内核操作系统的代表，不仅具有微内核操作系统的优点，而且经过对 IPC 机制的改进，其性能也十分出色。本论文选用了 L4 微内核操作系统规范中使用最广泛的 Fiasco 作为实现的基础，将通过对 Fiasco 的设计和源代码的改进和扩展来实现一个支持多核架构的微内核操作系统。

经过对 L4 规范和 Fiasco 源代码的分析[4][5]并阅读了国外其他学者对于多处理器架构上的 L4 微内核操作系统设计的研究。本文提出了自己的设计方案。针对嵌入式应用的场景和微内核操作系统的内核很小的特点，本文提出将分离模型应用到操作系统的整体设计中，也即在处理器的每一个核心上运行一个操作系统内核，多个内核之间相对独立但也有机制来进行同步操作。随后本文分别对操作系统的各个组成部分：内存管理、线程间通信、调度机制以及锁和同步机制进行了设计。对于内存管理，主要考虑的是内存在多个核之间有多大程度的共享，最终本设计选择了地址空间可以跨核存在并且保存地址空间映射关系的树型结构只存在一个的设计。对于线程间通信，为了充分利用多核处理器架构共享 L2 Cache 的特点，本文设计了一个邮箱系统作为多核间的 IPC 机制并采用代理线程的方式来实现透明性。对于调度机制，本设计将细粒度的调度限制在了处理器核心本地，并提供了线程迁移机制使得线程可以从一个处理器核心转移到另一个处理器核心。对于多核间的同步，本设计提供了自旋锁，IPI 和利用邮箱系统的选择。

在完成了系统的设计方案之后，在 Fiasco 代码的基础上编程实现之前的设计，包括对 Fiasco 原有的代码的修改和增加新的模块。在代码修改完成后，对整个系统的性能作全面的测试，评估其优劣。

1.5 软硬件平台

在软件方面，如果从头开始设计并实现一个完整的支持多处理器架构的微内核操作系统，工作量非常巨大，也完全没有必要，学术界已有许多开源的微内核操作系统，且允许用户对其进行修改并增加新的功能。因此本设计采用的是利用开源的微内核操作系统，在其基础上进行修改和扩展，以实现对多核处理器架构的支持。经过调研和比较，本设计最终选用了德国 Dresden 科技大学开发的 Fiasco 作为研究基础，Fiasco 作为 L4 规范的最流行的一个实现，功能完善并且版本稳定，非常适合于学术研究。而 L4 作为第二代微内核操作系统的代表，不仅具有微内核操作系统的诸多优点，而且经过对 IPC 的特别改善，性能方面也十分出色。在开发工具上，本设计使用 Linux 作为开发平台，并使用 vi + doxygen 作为开发工具，gcc 作为编译工具，svn 作为版本控制工具。

在硬件平台上，本设计在初期使用 VMWare 虚拟机调试。由于开发涉及底层部分，使用真机不断重启效率很低，而 VMWare 不仅能完整得模拟出双核处理器的硬件架构，而且启动很快，能大大提高调试速度，而且 VMWare 提供的录像功能非常适用于调试微内核操作系统的启动部分。在测试阶段，为了在实验硬件平台上测试微内核操作系统的性能，本文使用了基于 Intel Core2 Duo Q965 芯片组的开发平台。

1.6 论文结构

本文共分八章。除本章绪论外，之后各章安排如下：

第 2 章 多核架构和微内核操作系统介绍

首先比较了多处理器架构和多核架构，指出两者最大的区别是是否共享 L2 Cache。接着比较了微内核操作系统和单内核操作系统的区别，列举了微内核操作系统的优点。随后介绍了本论文所采用的 L4 微内核操作系统规范以及其实现 Fiasco。

第 3 章 总体架构设计

首先介绍了 L4 微内核的内核功能和系统调用。随后分析了采用全局模型来实现多核架构支持的问题。最后阐述了本论文所使用的分离模型并列举了其在嵌入式环境的应用场景。

第 4 章 线程间通信

介绍了 L4 的 IPC 机制分类及流程。随后分析了直接在原有 IPC 机制上修改会产生的问题。最后阐述了基于邮箱系统的新的跨核 IPC 机制。

第 5 章 系统功能设计

本章分别叙述了适应多核处理器架构的 L4 操作系统在内存管理、线程调度、同步机制和服务应用程序方面的设计方案。

第 6 章 系统功能实现

本章着重分析了新设计的 IPC 邮箱系统实现和性能分析。并给出了 IPC 系统的测试数据。

第 7 章 总结与展望

对本文作了简要的总结，并提出若干有待深入研究的问题。

第2章 多核架构和微内核操作系统介绍

2.1 多处理器架构和多核架构的比较

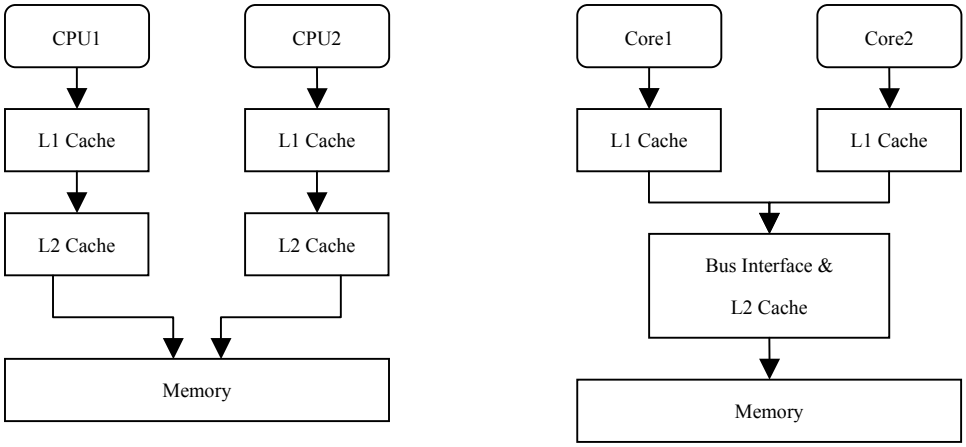

图 2-1 多处理器架构（左）和多核架构（右）的比较

Fig. 2-1 Comparison between SMP architecture (left) and multi-core architecture (right)

如前文所述，已经有不少的国外学者研究了支持多处理器架构的 L4 操作系统的设计，但由于多核架构和多处理器架构上的一些不同点，两者之上的 L4 操作系统的设计理念有非常大的区别。本文首先将分析多核架构和多处理器架构的不同之处。

如图 2-1 所示，多核架构的两个核共享一些电路，比如二级缓存、前端总线等。由于两个核在同一片芯片内部，互相之间的距离较近，这就允许缓存的同步电路运行在一个相当高的时钟频率上，这比多处理器架构需要信号在两片芯片之前传递的情况要快得多，而更高的时钟频率就使得单位时间里可以发送更多的数据。

另一方面，多处理器架构也有自己的优点，正因为两个处理器不共享前端总线，当两个处理器同时访问内存中不同地址的时候不会相互影响，这就相当于多处理器架构在访问内存的时候带宽要比多核架构大得多。但多核架构在两个核之间高速通信的优点在大多数情况下都能弥补它由于共享前端总线而造成的访问内存的性能上的损失。所以当前多核架构更加流行一些[6]。

综合以上所述，本文在设计支持多核架构的 L4 操作系统时，会放宽对多核间 IPC 的限制，因为系统可以利用共享的 L2 Cache 存放消息来实现高速的多核间 IPC。但考虑到共享前端总线，在设计时会避免频繁地访问内存。

2.2 微内核操作系统

2.2.1 微内核操作系统架构

所谓微内核是一种最小的计算机操作系统内核，其设计思想是内核本身不提供操作系统的相关服务，而是提供实现这些服务的机制，诸如底层的地址空间管理，线程调度以及进程间通信。一般的硬件都有执行权限级别，比如 Intel IA32 架构有 Ring0 到 Ring3 的特权级，而一般的操作系统至少会用到两个，称之为内核态和用户态。而对于微内核操作系统而言，其操作系统服务并不都运行于内核态，仅提供服务实现的机制部分运行于内核态；这里的操作系统服务，包括设备驱动、文件系统和用户界面等则作为用户态的服务应用程序运行。

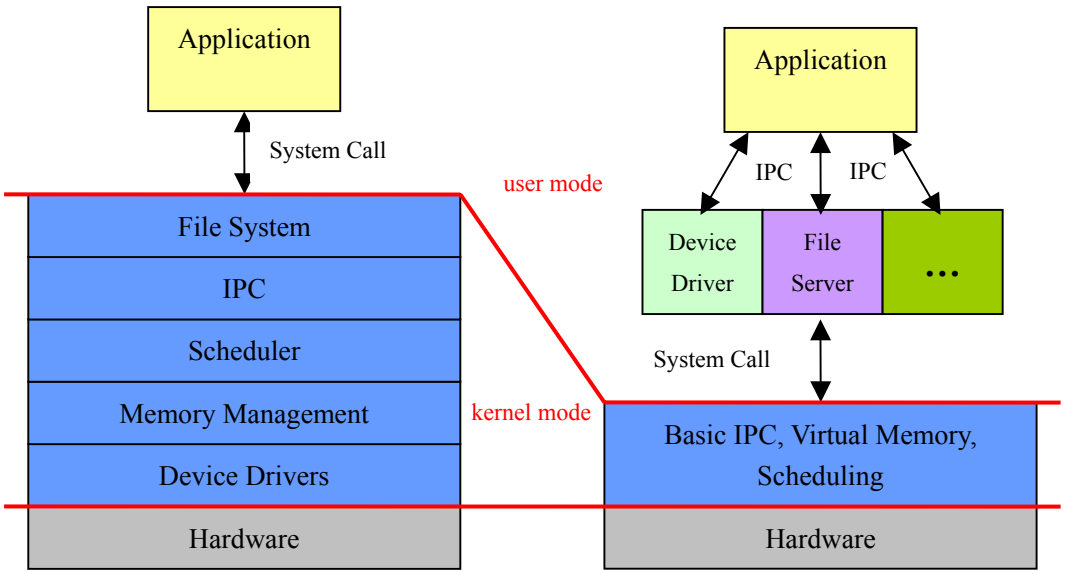

图 2-2 单内核操作系统（左）和微内核操作系统结构比较图

Fig. 2-2 Structure diagram of monolithic kernel (left) and micro-kernel operating system

如图 2-2 所示，左边的是单内核操作系统的结构示意图。常用的 Windows 操作系统和 Linux 操作系统都属于该类型。单内核操作系统的内核提供完整的操作系统服务，如图 2-2 中所示的文件系统、进程间通信、进程调度器、内存管

理、设备驱动程序等。而应用程序则运行在用户态，应用程序想要使用操作系统提供的服务，需要通过内核提供的接口，这被称之为系统调用。当应用程序进程执行系统调用时，会陷入到内核的代码去执行，进程此时运行于内核态，处理器处于特权级最高的 Ring0 状态执行。当应用程序需要的操作系统服务调用完成，进程回到原来的应用程序代码中继续执行，同时也回到了用户态，在 Ring3 的特权级上运行。

而对于右边的微内核操作系统，操作系统的内核部分被大大简化，只包括基本的 IPC 机制，虚拟内存映射和调度机制，这些仅仅是用于实现操作系统服务的基本机制。真正的操作系统服务，比如设备驱动、文件系统、应用程序间通信等，通过用户态服务程序的方式实现。当普通的应用程序需要操作系统的相关服务时，通过发送 IPC 消息给这个服务程序，这些服务程序进行相关的操作，必要时也会通过内核提供的系统调用陷入到内核态去执行基本的操作，并把结果再通过 IPC 返回给请求服务的应用程序。

2.2.2 微内核操作系统优点

微内核操作系统特有的架构带来了很多优点，这些优点正好符合嵌入式平台对操作系统的需求，非常适合于嵌入式环境的应用。

首先是可靠性，按照单内核操作系统的设计，内核包括所有的操作系统服务，其中任何一个服务出错，就会造成整个系统的崩溃[7]。微内核操作系统的设计思想是在内核中留尽量少的东西，只保留实现操作系统服务的最基本机制，而把具体服务的实现放到用户态的服务应用程序中去。这就大大降低了内核崩溃的几率。特别是目前操作系统的许多错误都是因为不规范，并且没有经过严格测试的驱动程序造成的。图 2-3 显示了 Linux 内核代码不同目录的错误分布，显然驱动程序的错误是主要原因。

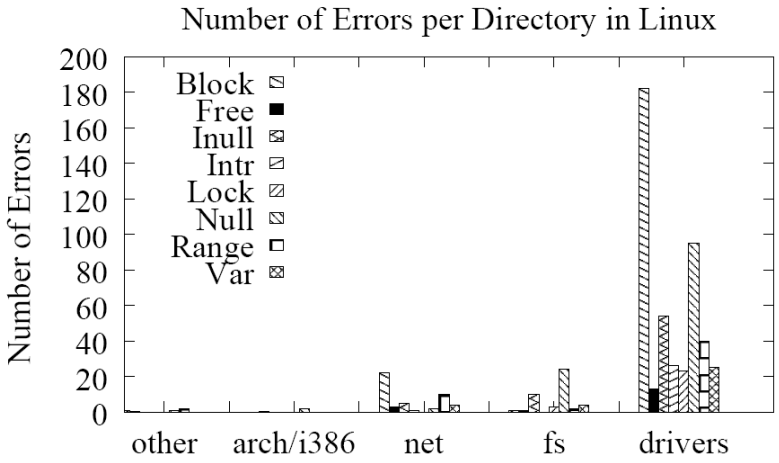

图 2-3 操作系统中的错误在各模块中的分布（来源：文献[8]）

Fig. 2-3 Distribution of operating system's errors in different modules (Source: Reference[8])

然后是实时性。实时性是指操作系统对响应时间有严格的要求，不仅要求成功执行还要求在规定的时间内完成所有操作。一般会要求操作系统内核是抢占式的，并且内核的代码是可重入的[9]。微内核操作系统内核较小也意味着需要实时化的部分较少。而且微内核之上也可以很方便同时运行实时的程序和一般的非实时的程序。

最后是安全性。设计安全系统的准则是最小权限准则，也即所有的部件都只拥有执行相关功能所必须的权限，而没有额外的权限。最小权限准则需要系统的可信计算基础尽可能地小[10]。可信计算基础（Trusted Computing Base）是指整个系统中对安全性最关键的部分，在可信计算基础内的漏洞会危及整个系统的安全性。因为操作系统内核是可信计算基础中最重要的一部分，微内核操作系统内核较小的特点使之在安全性应用方面很有优势。

上面说了很多微内核操作系统的优点，但微内核架构也不是完美的，它有一个很大的缺点那就是性能。对于单内核操作系统，调用系统服务的方式是通过系统调用，需要的仅仅是用户态和内核态的两次转换，每个进程都同时有用户栈和内核栈，可以存放执行过程中的信息。而对于微内核操作系统，调用系统服务需要通过发送 IPC 消息给服务应用程序，服务应用程序通过系统调用完成服务请求后再通过另一个 IPC 消息把结果返回给调用者。这涉及到进程的上下文切换，并且由于没有内核栈这样简单的机制，传送消息需要额外的拷贝开销。因此性能对于微内核架构的操作系统就成了一个很大的问题。事实上第一代微内核操作系统

诸如 Mach 之上运行的系统的性能的确让人难以接受。然而以 L4 为首的第二代微内核操作系统通过合理的架构设计，将 IPC 的开销相对 Mach 成数量级地减少。图 2-4 所示的分析表明在 L4 微内核操作系统之上运行的 L4Linux 的性能相对原本的 Linux 只有几个百分点的损失。

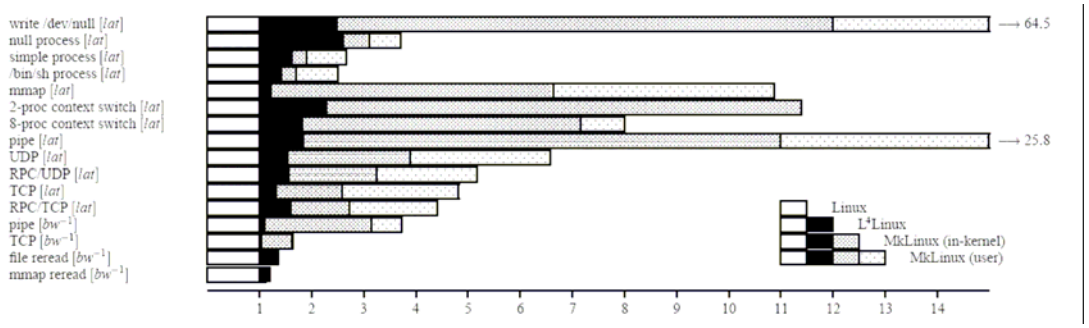

图 2-4 Linux 和 L4Linux 性能比较（来源：文献[11]）

Fig. 2-4 Performance comparison between Linux and L4Linux (Source: Reference[11])

2.3 L4 规范

第一个 L4 微内核操作系统由 Jochen Liedtke 开发[12]，并在 Intel IA32 架构上实现。如前文所述，Mach 为首的第一代微内核操作系统的性能不尽如人意，Jochen Liedtke 经过分析发现造成 Mach 性能问题的关键因素是 IPC 机制过于复杂。他开始着手设计一个全新的微内核操作系统规范，并把性能作为首先考虑的因素。于是产生 L4 微内核操作系统规范。L4 微内核操作系统规范的 IPC 机制简单并且高效，使得 L4 微内核操作系统和诸如 Mach 的第一代微内核操作系统相比性能有了很大的提升。然而 Jochen Liedtke 发现还可以通过把一些机制移出操作系统内核的方式进一步提高 IPC 的效率，于是便有了 L4 规范。L4 微内核操作系统被称为第二代微内核操作系统。

最初的 L4 微内核操作系统规范是指 L4.V2 规范。通常所说的 L4 规范默认就是 L4.V2 规范。然而 L4.V2 规范也有一些问题，比如线程 id 号使用起来不够灵活，此外，规范提出的部落和首领概念效率不高，并在实际中很少被采用。于是便有了 L4.X0 规范来改进这些问题。但这个版本规范的主要目的并不是解决之前的问题，而是作为一个测试用例实验，这也是规范名称中 X 的由来（eXperimental）。在随后的 L4.X2 规范中，设计上作了很大的改变，来解决 L4.V2

规范带来的问题。主要的改动包括：任务（task）和线程（thread）管理的系统调用被分成了很多个，支持多进程处理和将应用程序接口（API）和二进制程序接口（ABI）分离得更加清楚。有时 L4.X2 规范也称为 L4.V4 规范。此外还有 Dresden 科技大学的 Kauer and Volp 所开发的 L4.Sec 规范，这个规范侧重于 L4 的安全性方面的特性。

表 2-1 L4 各实现版本比较（来源：文献[13]）

Table 2-1 Comparison between various L4 implementations (Source: Reference[13])

名称	CPU	开发语言	许可证	开发机构
Pistachio	奔腾及以上, IA64, PowerPC, Alpha, 64-bit MIPS	C++	BSD	L4KA at Uni KA and UNSW
Fiasco	i486 及以上, StrongARM, Linux	C++	GPL 或商业	TU Dresden
P4	x86, MIPS, PowerPC, ARM	C	商业	Sysgo AG
OKL4	x86, MIPS, ARM	C++	BSD 或商业	OK Labs
L4 for PPC	PowerPC 603e	C	不提供	Univ. of York
Hazelnut	Pentium 及以上，StrongARM	C++	GPL 或商业	L4KA at Uni KA
L4/MIPS	MIPS R4x00	C	GPL	UNSW
L4/Alpha	Alpha AXP 21264	汇编	GPL	TU Dresden, UNSW
L4/x86	i486 or better	汇编		GMD, IBM Watson, Uni KA

此外 L4 也有很多实现版本，如表 2-1 所示。比如最常见的 L4.Fiasco，偏重性能的 L4Ka::Hazelnut，偏重移植性的 L4Ka::Pistachio，偏重安全性的 L4.sec，以及应用于嵌入式系统的 Pistachio-embedded 等等。本论文使用的是 Fiasco 微内核操作系统。

2.4 Fiasco

Fiasco 微内核操作系统是由 Dresden 科技大学的 Michael Hohmuth 等人开发的。最初 Dresden 科技大学想要开发 DROPS 操作系统。DROPS 的系统结构如图

2-5 所示。DROPS 操作系统是一个分布式的实时操作系统，并且基于 L4 规范开发。然而 Michael Hohmuth 等人发现由 Jochen Liedtke 开发的 L4/x86 实现时有一些严重的缺陷，包括代码的可读性、可维护性以及协议证书等问题。于是 Dresden 科技大学决定自己开发他们自己的 L4 规范实现，并命名为 Fiasco，并在此之上搭建 DROPS。

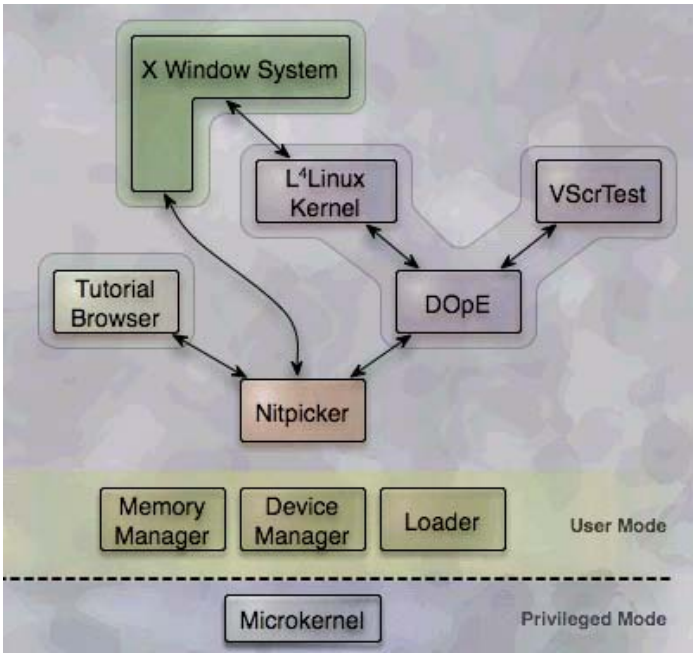

图 2-5 DROPS 操作系统层次图（来源：文献[14]）

Fig. 2-5 DROPS system hierarchy diagram (Source: Reference[14])

Fiasco 微内核操作系统主要基于 L4.V2 规范开发，这是因为 L4.V2 规范不仅稳定而且拥有 L4 的所有完整的特性，而 L4.X2 等还仅是实验版本。Fiasco 最初在 IA32 架构上实现，随后被移植到 IA64 和 ARM 架构上，此外 Udo Steinberg 还开发了可以在 Linux 用户态运行的 Fiasco-UX。作为第二代微内核操作系统，Fiasco 的内核非常小。相比 Linux 的 320 万行代码，Fiasco 内核的代码只有 20000 行左右。Fiasco 主要由 C++和汇编语言写成，汇编部分主要用户实现 C++所无法触及的底层操作或者是基于一些优化的目的。此外 Fiasco 代码还用到了一个用 Perl 语言实现的预编译器，其目的一方面是处理不同平台的多态性，可以在编译时判断目标平台，并仅链接对应平台的代码，这就避免了在程序中判断平台并进行分支的开销；另一方面预处理器允许编写 C++函数的声明和定义写在同一个文件里，由预处理器来分成声明部分和定义部分并建立包含关系。

此外 Fiasco 微内核操作系统的一大特征是其对实时性的支持。Fiasco 是支持硬优先级的抢占式实时操作系统。它使用非阻塞的方式实现内核部件的同步，保证了优先级的继承并使可运行的高优先级线程永远不会被低优先级进程阻塞。当在 Fiasco 之上运行 L4Linux 时，硬实时的应用程序可以和一般的 Linux 应用程序共享系统资源。Fiasco 作为第二代微内核操作系统，由于其以性能为主导的设计思想带来的高效的上下文切换机制，在将应用程序保护在地址空间中时，由地址空间保护带来的性能惩罚几乎可以忽略不计。这方面的性能大大超越第一代操作系统。

这因为 Fiasco 这多方面的优点，它成为了 L4 的各个实现中在学术界和工程界应用最广泛的一个。本论文的研究也将在 Fiasco 的基础之上进行修改和扩充。

2.5 本章小结

本章分析了多处理器架构和多核架构的具体区别，并针对是否共享 L2 Cache 这个最大的区别提出了本设计的支持多核架构的微内核操作系统的设计思路。随后介绍了微内核操作系统的基本概念，并将微内核架构和一般的单内核架构作了比较。微内核架构的内核部分非常较小，操作系统所需的服务并不全部实现在内核内，内核仅提供实现的基本机制，而把服务的实现放到了用户态的应用程序中。接着分析了微内核操作系统的特点，其在可靠性、实时性和安全性方面优势明显，而这些优势恰恰是嵌入式环境所需要的。最后介绍了本论文使用的 L4 微内核操作系统规范和 L4 操作系统的一个具体实现：Fiasco。

第3章 总体架构设计

3.1 L4 系统架构

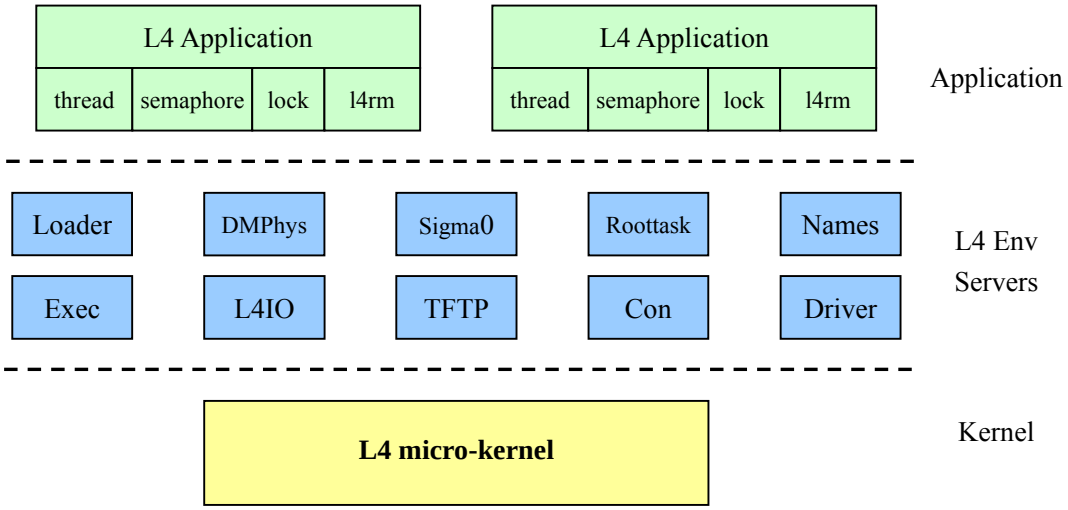

图 3-1 L4 系统架构层次图

Fig. 3-1 L4 system hierarchy diagram

L4 作为第二代微内核操作系统的代表，其系统架构非常符合微内核操作系统的设计理念。为 L4 的系统架构层次图，层次大致可以分成内核、服务应用程序和具体的应用程序三层。在操作系统的内核部分，L4 只提供了三个最基本的机制，作为实现具体操作系统服务的基础。这三个机制是：基本的内存管理、线程调度和线程间通信[15]。

3.1.1 L4 内核功能

基本的内存管理

基本的内存管理包括几个部分。比如虚拟地址到物理地址的映射，在 L4 中采用对内存的映射有两种方式：对于内核态的地址空间使用线性偏移的方式来获得虚拟地址到物理地址的映射；而对于用户态的地址空间采用一般操作系统标准的段页式模式来映射，对于页表的查询也需要内核来完成。此外 L4 的内核还提供地址空间的授予、映射和收回操作，并且提供映射树来记录地址空间之间的授

予、映射和收回关系。但是内核仅仅提供的是操作的机制，内核本身并不负责对所有的内存进行管理，对于内存的分配管理都是通过运行在用户态的服务应用程序 Sigma0 和 Roottask 来完成的。

线程调度

在 L4 中没有进程的概念，取而代之的是任务和线程的概念。一个任务可以包括多个线程，每个任务对应了一个地址空间，而线程是 CPU 调度的基本单位。L4 的内核提供了完整的线程调度功能。L4 的调度基于静态优先级，当当前任务完成或者用完了自己的时间片的时候，内核在就绪队列中找到优先级最高的队列占用 CPU，当有多个线程处于这个最高的优先级时，它们会使用 Round-Robin 的模式共享 CPU 轮流执行。不仅如此，L4 作为一个支持实时性的抢占式操作系统，还支持强制调度，内核提供机制当发现有比当前正占据 CPU 的线程的优先级更高的线程时，会发生强制调度。

线程间通信

线程间通信就是 IPC（Inter Process Communication 进程间通信）。由于 L4 的执行单元是线程，因此变成线程间通信。然而 IPC 是一个约定俗成的术语，在本文的后面部分会继续使用 IPC 来指代 L4 的线程间通信。L4 提供了 send, receive, wait, call, reply-wait 五种 IPC，并且还可以按 IPC 消息的储存媒质分为 Short IPC 和 Long IPC。但这先 IPC 机制使用一个统一的接口来向外提供服务。由于微内核操作系统的特点是应用程序调用操作系统的服务时需要频繁地通过 IPC 来向服务应用程序发出请求，因此 IPC 的性能对于微内核操作系统来说非常关键。L4 对于在设计时非常重视性能问题，因此对 IPC 机制作了很多优化，其 IPC 机制的效率在微内核操作系统中属于很高的。

3.1.2 L4 的系统调用

服务应用程序或者一般的应用程序需要调用 L4 的内核提供的机制需要通过系统调用，L4 内核一共向外提供了 5 种系统调用。

ipc

ipc 系统调用用于实现线程间的 IPC。IPC 需要传递的消息可以直接放到寄存器中，或者把消息放到接收线程的地址空间中，而把对应内存的地址放到寄存器中。所有的 IPC 操作都是同步并且没有缓冲，发送线程发起 ipc 系统调用后进入等待状态，直到接收线程也发起 ipc 系统调用，两边达到握手状态，完成消息的发送，IPC 结束。ipc 系统调用除了用来传递消息，还可以用来实现地址空间的授予和映射，此时 fpage 字就作为需要传递的消息。

id_nearest

如果在发起 id_nearest 系统调用的时候，在寄存器 ESI 和 EDI 中放了关键字 nil（0xFFFFFFFF），系统调用将会返回当先线程的 id 号。而如果在 ESI 和 EDI 中指定了某个有效的线程 id，系统调用将会返回最接近这个 id 的线程 id 号，这种情况主要应用于在 L4 的 clan-chief 机制中。

fpage_unmap

地址空间的授予和映射操作都通过 ipc 系统调用实现，而回收操作则需要 fpage_unmap 系统
调用实现。调用 fpage_unmap 后，寄存器中指定的 fpage 将会从所有的地址空间中回收。

thread_switch

thread_switch 系统调用用于当前的线程完成操作，或者时间片用完主动放弃 CPU 的情况。系统将从就绪队列中找出优先级最高的线程占用 CPU。此外，后文将会提到的 L4 的时间片捐献机制也是通过这个系统调用实现的，在寄存器中指定想要捐献时间片的对象线程，再调用 thread_switch 系统调用即可。

thread_schedule

thread_schedule 系统调用主要用于修改一个线程的优先级，时间片和外部强制调度器等信息。此外也可以用于查看某个线程的这些相关信息。

lthread_ex_regs

lthread_ex_regs 系统调用主要用于查看和修改线程的 ESP，EIP 等寄存器的值。此外 thread_ex_regs 系统调用也可以用于创建新线程。

task_new

task_new 系统调用用于创建或者删除一个任务。删除一个任务意味着和这个任务相关联的所有地址空间，以及这个任务所包括的所有线程都会同时被消除。

3.2 全局模型

对于多核上的操作系统设计，比较容易想到的，同时也是应用比较普遍的是全局模型。在全局模型中，操作系统内核只有一个实例，运行在某一个核上（一般称为 BSP, Bootstrap Processor）上。在这种模式下，CPU 核心对于操作系统仅仅是一种系统资源，操作系统内核上运行的调度器（对于微内核操作系统来说，调度器也可以运行于用户态，但效率会下降很多）负责把线程分配到指定的 CPU 上。

对于全局模型可以选择为每个处理器核心都维护一个就绪线程链表或者维护一个全局的就绪链表。这两种设计各有所长，前者适用于多个线程在一个处理器核心上轮流执行的场景，可以减少调度器判断并选择处理器核心的次数，但是这种设计很容易造成多核负载的不平衡。需要调度器经常查看各个核的状态，必要时进行负载平衡。而后一种设计不会有负载平衡的问题，但是每次某个线程时间片用完，回到就绪队列，下次又需要为这个线程再找一个处理器核心来执行，效率方面会有所下降。

如图 3-2 所示为全局就绪链表的情况。当某个核上运行的线程执行完毕或者用完了自己的时间片，会通知操作系统内核中的调度器，调度器从就绪链表中找到优先级最高的线程，将其调度到刚才的处理器核心上运行。还有一种情况是就绪链表为空的情况下，新产生了一个线程希望被执行。调度器会一一查看所有处理器核心的状态，如果能找到一个空闲的处理器，则把线程调度到该处理器核心上运行，否则则把线程加入就绪链表。

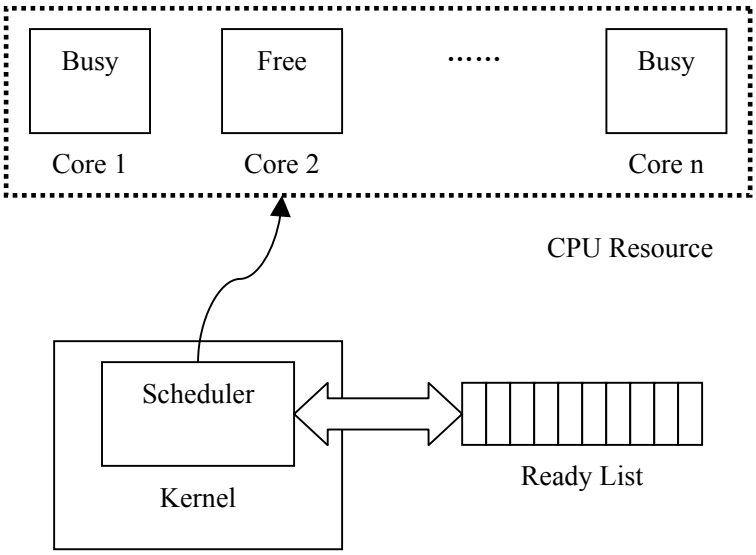

图 3-2 全局模型示意图

Fig. 3-2 Global model diagram

显然这种全局模型的调度机制和原本的 L4 机制区别很大，需要大量修改原有的 L4 代码，并且系统结构复杂，难以保证微内核操作系统应有的效率，因此本论文没有采用这种全局模型方案。

3.3 分离模型

3.3.1 离散模型架构

首先为避免混淆，在此明确两个术语的含义。对于多核的“核”（core），本文使用处理器“核心”或者“核”这个称呼，对于微内核操作系统的“核”（kernel），本文使用“内核”这个称呼。

正因为 L4 是微内核架构的操作系统，本设计采用另一种选择，就是在每一个处理器核心（Core）上运行一个 L4 操作系统的内核（Kernel)，称之为分离模型。分离模型架构的示意图如图 3-3 所示。

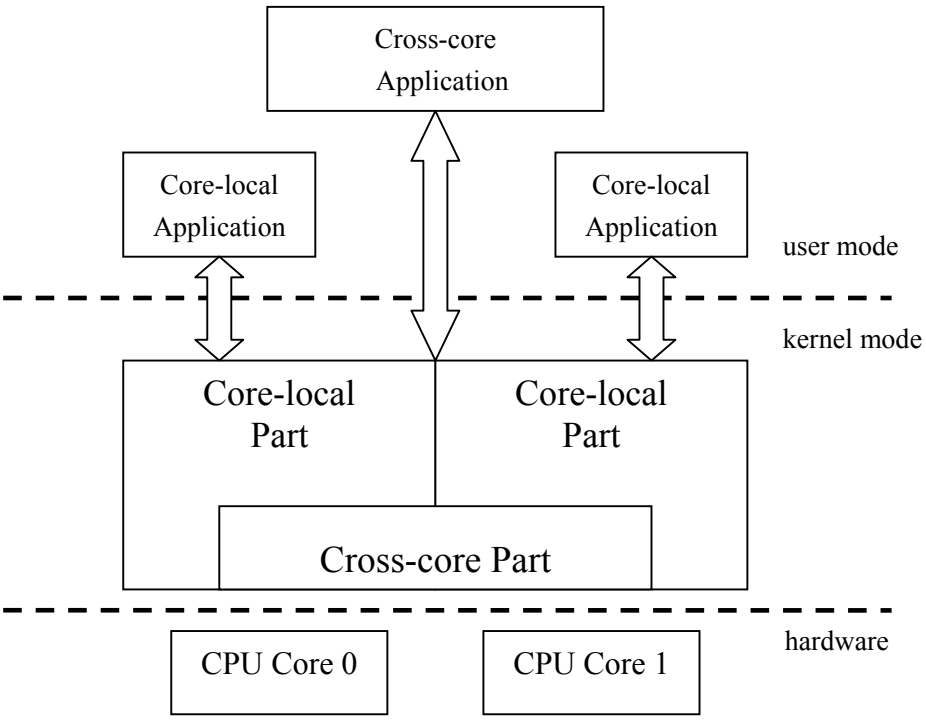

图 3-3 离散模型结构示意图

Fig. 3-3 Split model structure diagram

本设计可以采用分离模型的架构设计的前提正是因为 L4 作为微内核操作系统，其内核部分的尺寸很小，启动后仅占用很小的内存，不会因为每个处理器核心上都运行系统内核，内核占用的内存成倍增加而占用大量内存，使其它应用程序可用的内存大量减少。

分离模型的一大优点是可以充分利用原有的代码。因为原本已经有了支持单处理器硬件平台的 Fiasco 的代码。将微内核进行复制，使之运行在每个处理器核心上所需要的对代码的修改较少，主要的工作量在于建立一个多核间通信的机制和修改操作系统需要在多核间共享的部分。这相比于对原有的支持支单处理器硬件平台的代码进行修改，使所有的机制和服务都支持多核处理器平台来说，工作量要小很多。

分离模型另一大优点是提供了很好的隔离性。将操作系统内核复制，并运行在多个处理器核心上，在内核上运行的线程自动被限制在了操作系统内核所处的处理器核心上。L4 原本的调度机制可以作为处理器核心内部的调度，而设计时只需提供额外的跨核操作和调度机制即可。

当然在每个处理器核心上跑的操作系统内核之间也不可能完全独立，一方面，系统的资源比如内存，硬盘和各种 I/O 等必须共享，因此需要同步的机制。另一方面，多核之间也需要进行通信，以及合作进行并行计算等工作，所以需要跨核的机制提供这些服务。因此，本设计的微内核操作系统可以分成两部分：处理器核心独立的部分，这部分系统主要继承自原本的支持单处理器硬件平台的 L4V2 规范的 Fiasco 实现并进行少量的修改，功能上主要负责处理器核心本地的操作，比如处理器核心本地的调度，处理器核心本地的线程间通信等等；而另一部分则是跨处理器核心的部分，这部分主要基本上是新添加的设计，主要是出于不同处理器核心上的线程间通信，系统共享资源的同步机制等等。

如图 3-4 所示为基于离散模型的微内核操作系统层次结构示意图。整个系统大致可以分成硬件层，内核层，服务层，应用层四个层次。

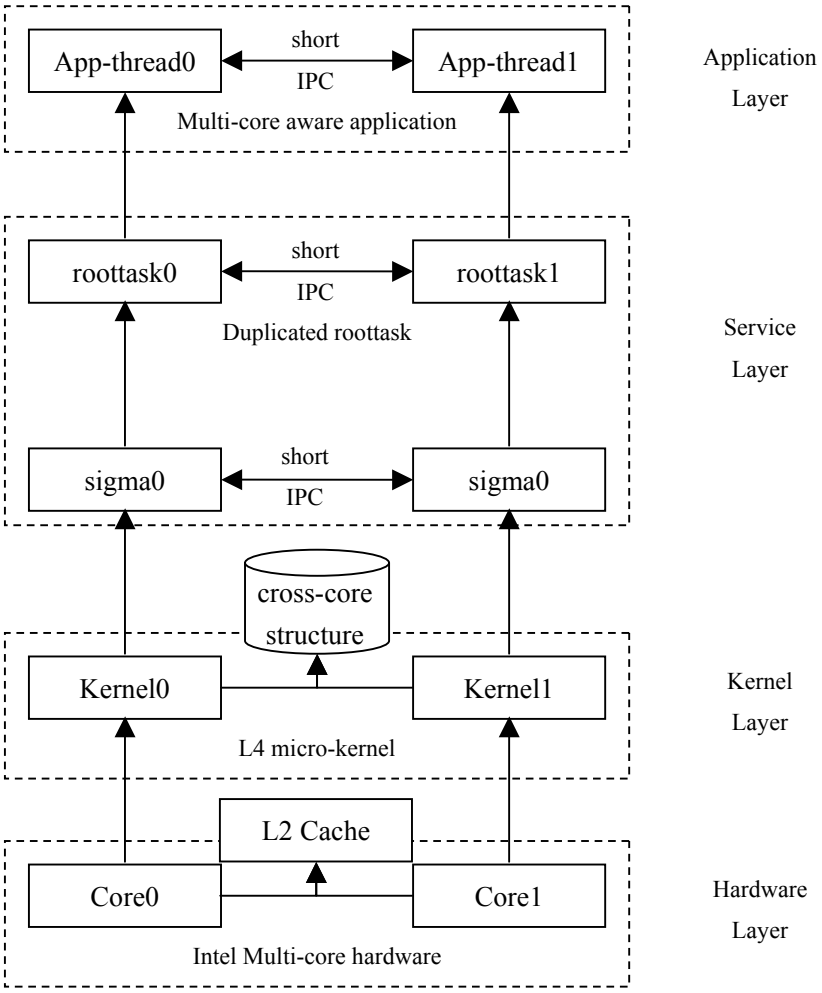

图 3-4 离散模型层次示意图

Fig. 3-4 Split model hierarchy diagram

硬件层

最底层的是硬件层，多核处理器架构的每一个处理器核心上会运行一个操作系统内核，内核独占这个处理器核心以及配套的寄存器，L1 Cache 以及 APIC 等，而其它硬件资源，诸如内存，硬盘，I/O 等则被所有的内核共享，特别需要注意的是多核处理器架构中的 L2 Cache 被所有内核共享。

内核层

硬件层之上的是操作系统内核层，主要提供用以实现操作系统服务的底层机制。内核可以分成内核本地的部分和跨内核的部分，前者负责在处理器核心本地操作的机制，如本地的线程间通信和本地的线程调度等等，而后者负责跨核的操作，如地址空间的管理，跨核的线程间通信等。

服务层

在内核层之上的是服务层，微内核操作系统的操作系统服务，如内存的分配，驱动程序等都是通过运行在用户态的应用程序实现的。服务应用程序通过调用内核提供的系统调用来实现操作系统的各种服务。这些应用程序可以以多种模式运行在内核层之上，具体模式将在后文详述。L4 内置了两个必须的服务应用程序，分别为 sigma0 和 roottask。sigma0 是默认的内存分配器而 roottask 是系统硬件资源的管理器。

应用层

在服务层之上的则是应用层，运行各种执行特定功能的应用程序，并通过 IPC 调用服务层的服务应用程序获取操作系统的服务。应用程序本身也可以有本地模式，主从模式和离散模式多种选择。

3.3.2 分离模型应用场景

基于离散模型的架构设计的优点不仅仅是可以大量复用原有的 L4 代码，它还非常适用于嵌入式场景。

可靠性

比如嵌入式设备需要高可靠性，本设计可以使用一个内核运行所有的应用程序，而另一个内核作为热备份。由于两个核上都运行了操作系统，就算操作系统崩溃，也可以较为平滑得过度到热备份的核上。

实时性

此外，L4 是一个实时的操作系统，本设计可以指定一个核的资源完全用于实时的应用程序，而另一个核的资源留给非实时的程序，这样的隔离架构非常清晰，而实现方式只要指定一个操作系统内核运行在实时模式，而另一个运行在非实时即可。

安全性

不仅如此，离散模型在安全性上也有应用，L4 的一个著名的应用是在微内核上运行多个 L4Linux，密钥放在一个 L4Linux 中，而执行程序运行在另一个中。而现在本设计把密钥放在某个核上运行的 L4 内，而程序执行在另一个核的 L4 之上，实现更加彻底的隔离。

3.4 本章小结

本章首先分析了 L4 微内核操作系统的系统架构，并详细说明了 L4 微内核操作系统在内核中的三大机制：基本的内存管理、线程调度和线程间通信以及五大系统调用各自的功能。接着分析了将 L4 微内核操作系统应用到多核处理器架构上的方案，分别详细描述了全局模型和离散模型方案以及各自的优缺点。基于微内核和嵌入式设备的特点，本设计选用离散模型作为设计方案。最后阐述了一些基于离散模型的微内核操作系统在嵌入式环境中应用的场景。

第4章 线程间通信

4.1 L4 的 IPC 机制

4.1.1 IPC 分类

IPC 是指 Inter Process Communication 进程间通信，对于 L4 来说其实是线程间通信，但本文保留惯用的称呼。

L4 提供 5 种不同的 IPC，分别是：send, receive, wait, reply-wait 和 call。发起 send IPC 的线程只负责把消息发出，然后继续从事其他工作。对应的发起 receive IPC 的线程进入等待状态，直到从任意的发送线程接收到 IPC 消息。wait IPC 调用和 receive IPC 比较接近，但对发送线程有要求，只接收来自指定发送线程的消息。call IPC 调用相当于 send 和 wait IPC 调用的组合，线程首先发出消息，然后进入等待状态，直到它发送消息的线程发送回复消息给它。而 reply-wait IPC 调用相当于 send 和 receive IPC 调用的组合，线程发送消息，然后等待来自任意线程（不必是它发送消息的线程）的返回消息。

call IPC 机制的 send 和 receive 两个阶段在同一个 IPC 中的设计大大提高了效率。线程调用 IPC 的方式是通过系统调用，而系统调用涉及到了用户态和内核态的转换，开销是比较大的。将线程间通信合并到尽量少的系统调用中无疑能提高系统的效率。此外系统还可以把 send 和 receive 两个阶段的转换做成原子操作来进一步提高 IPC 的效率。

后两种 IPC 调用可以用在服务器的模式，服务线程总是处理来自客户线程的请求，然后查看是否有其它请求，这就是 reply-wait IPC 机制。而当客户线程和服务线程建立持续的连接，通过连续发发送接收传送数据的时候，使用的就是 call IPC 机制。

L4 的 IPC 也可以按照存放消息的媒介分成 Short IPC 和 Long IPC。Short IPC 使用寄存器来存放消息，所以消息的长度受可用寄存器的大小的限制。Short IPC 的好处是速度非常快，并且不会产生缺页错误。对于消息量很大的情况，可以使

用 Long IPC，Long IPC 使用内存来存放消息。在传送 IPC 消息时会产生一个 IPC 窗口，IPC 窗口处于 IPC 接收线程的地址空间中，并传送过程中被映射到了发送端的地址空间。发送线程直接把消息复制到接收线程的地址空间中，并把内存的地址放到寄存器中，发送 IPC 给接收线程完成消息的发送。这种不需要内核来专门开辟缓存的方式大大提高了 IPC 的速度。但是 Long IPC 因为需要访问用户态的地址空间，依旧有可能会产生缺页中断，这时候就需要把缺页信息发送给接收线程的 pager。因此平均来说，Long IPC 的速度还是要比 Short IPC 慢。还有一种特别的情况就是送一个字符串。此时系统首先会尽量把字符串使用 Short IPC 的方式放到寄存器里传送，对于无法放在寄存器里的部分再用 Long IPC 的方式传送。

IPC 机制还被用于处理硬件中断和软件中断，以及 map, grant 和 unmap 内存页。如图 4-1 所示，IPC 的接收端只可能是普通的线程，而对于 IPC 的发送端，除了普通的线程之外，还有两种伪发送端：IRQ 和 Preemption。当有硬中断或软中断产生时，系统需要通知目标线程。系统会把中断信息放到 IPC 的消息中，并把 IRQ 中断源伪装成 IPC 的发送端，通过 IPC 的方式把中断发送给目标线程。而 Preemption 对应系统的强制调度功能，实现方式也是通过 IPC 发送相关的消息给目标进程。IPC 还有超时机制，处于通信状态的 IPC 超时会造成 IPC 被取消[16][17]。

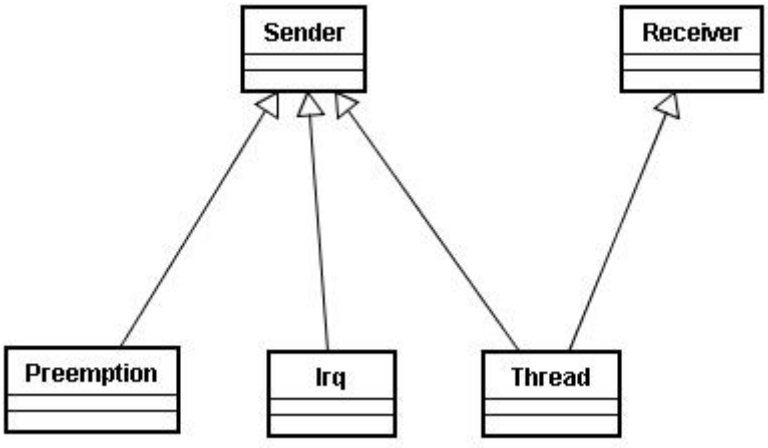

图 4-1 IPC 相关概念的继承关系图

Fig. 4-1 IPC object inheritance relationship diagram

4.1.2 IPC 流程

L4 的 IPC 机制是同步的，在发送端开始发送消息之前，发送线程和接收线程必须先达到握手状态。如图 4-2 所示，发送线程在准备发送消息前先会查看接收线程的状态，但接收线程未必处于就绪状态，该线程可能正在执行其它任务甚至正在和其它线程进行 IPC 通信的过程中。此时发送线程进入等待状态，系统会把该发送进程加入到接收线程的发送队列中。直到接收线程完成了其它任务，把发送线程从发送队列中取出，发送线程和接收线程同时进入握手状态。发送线程随后把消息发送给接收线程。在这个过程中如果出现错误，则整个 IPC 会被取消。

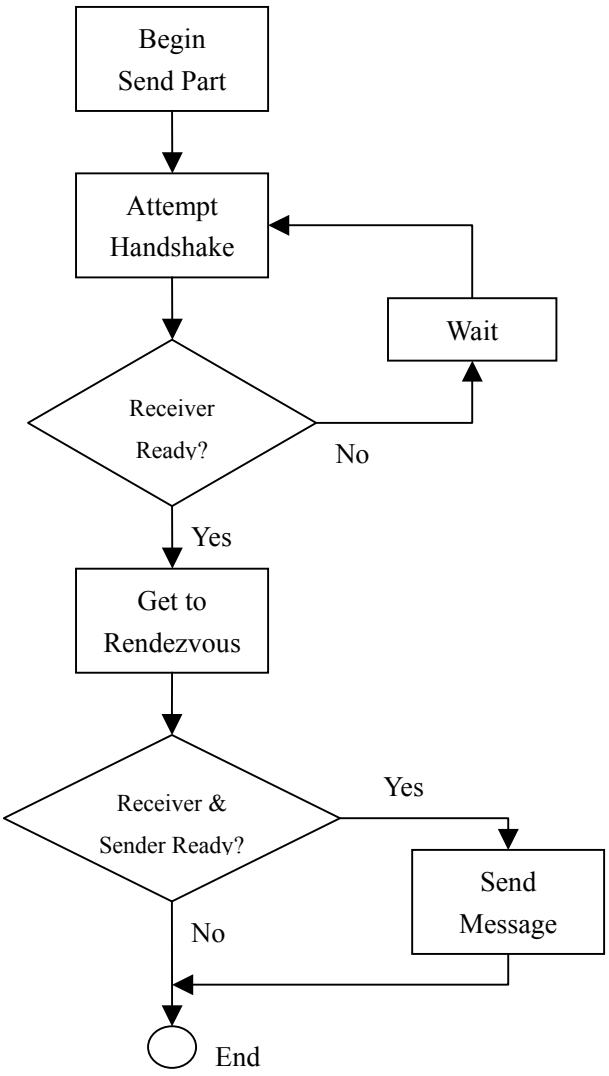

图 4-2 IPC 发送流程图

Fig. 4-2 IPC send procedure diagram

而对于 IPC 的接收部分来说，系统会为接收线程维护一个发送线程队列。图 4-3 为 IPC 接收部分，分为接收指定发送线程（wait IPC 和 reply-wait IPC 情况）和接收任意发送线程（receive IPC 和 call IPC 情况）两种场景。对于前面一种场景，当接收线程准备好进行 IPC 的时候，系统会找到接收队列中特定的发送线程，如果对应的线程存在，系统通知该线程发送消息，并把该线程移出发送线程队列。对于后一种场景，如果发送线程队列非空，系统通知队首的线程发送消息，并把它移出队列。当发送线程完成发送消息后，IPC 接收部分也随之结束。

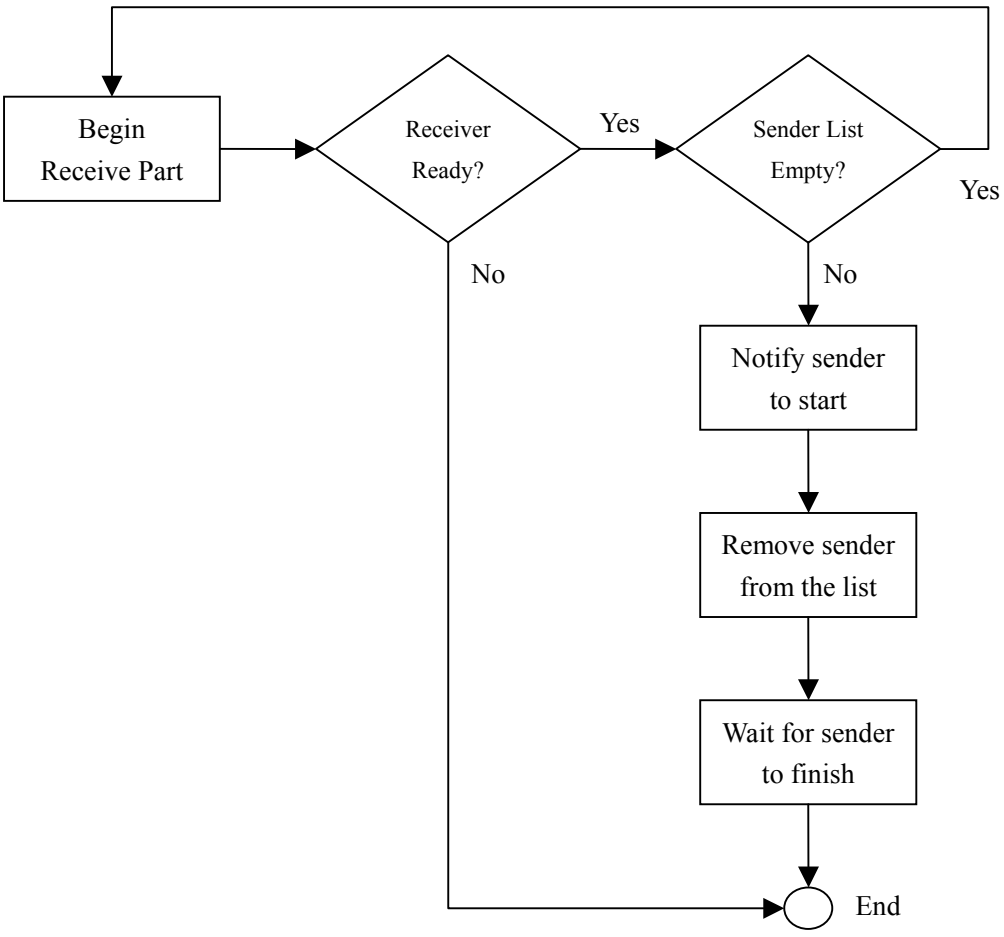

图 4-3 IPC 接收流程图

Fig. 4-3 IPC receive procedure diagram

4.2 多核间 IPC

在新的操作系统架构设计中，原本的 IPC 机制依旧可以用于处于同一个内核中（也即同一个处理器核心上）的两个线程间的线程间通信。但对于两个线程处于不同的内核中（也即处于不同的处理器核心上的情况），原本的 IPC 机制不能直接使用，需要设计新的 IPC 机制来应对这种情况。

一种设计方案是在原本的 L4 IPC 机制上进行修改，使之可以适应两个线程分处不同内核中的情况。但这种设计方案撇开设计复杂，修改工作量较大，还存在下面这些问题：

1. 首先原本 IPC 机制中的 short IPC 机制利用处理器的寄存器来传递消息。比如在 IA32 架构下，用到的是 EBX 和 EDX 寄存器，但对于两个线程处于不同处理器核心上的情况，两个处理器核心有各自的 EBX 和 EDX 寄存器，因此基于寄存器的消息传递机制完全不能起作用。

2. 其次原本的 IPC 机制是完全同步的，发送线程准备好后进入阻塞状态，直到接收线程从发送线程队列中选择了该发送线程，并进入就绪状态，两边完成握手，才开始发送消息。但这样的同步传递到了多核的情况下就会产生问题。发送线程和接收线程不在同一个处理器核心上，当发送线程进入阻塞状态后很可能处理器调度了其它线程占用 CPU，当接收线程进入就绪状态时会发现无法进行握手。

3. 此外原本的 IPC 机制中还有时间片捐献的机制。在 call IPC 或者 reply-wait IPC 中，当发送线程完成发送，需要调度到接收线程的时候，系统并不进行调度上下文的切换，而是直接在发送线程的调度上下文中运行接收线程。这样接收线程就可以先继续使用发送线程的时间片。但当发送线程和接收线程处于不同的处理器上时，接收线程没法直接使用发送线程在另一个处理器核心上的时间片，除非把发送线程迁移到接收线程所处的处理器核心上，但显然这样得不偿失，反而损失了更多的时间。

4. 对于微内核操作系统来说，IPC 的效率是首先要考虑的问题。由于基于寄存器的消息传递不能应用在跨核的情况，只剩下基于内存的消息传递可用。但是原本的 IPC 机制中的基于内存的消息传递使用接收线程的地址空间，每次 IPC 调用都要访问内存，效率会很低下。而接收线程的地址空间地址随机性很大，不太可能因为频繁地访问而被调入 L2 Cache 增加访问速度，因此也没有很好得利用到多核处理器架构共享 L2 Cache 的特性。

5. 最后，在本设计基于分离模型微内核操作系统中，发送线程和接收线程处于不同处理器核心上的 IPC 调用相对于发送和接收线程都在同一个处理器核心上只是少数情况，如果修改原有的 IPC 机制，使之适应跨核的情况，则 IPC 机制必须同时兼顾两种情况，这样虽然能提供很好的兼容性，但无疑会造成 IPC 机制的效率大大下降，特别是本地 IPC 的效率下降。这将严重影响整个系统的性能。

综上所述，本设计将会保留原有的 IPC 机制，但仅会应用在发送线程和接收线程处于同一个处理器核心上的本地情况。而对于发送线程和接收线程处于不同处理器核心上的情况，为此专门设计了一个邮箱系统。

4.3 邮箱系统

如前文所述，在 L4 原有的 IPC 机制上进行修改，使之支持跨核的 IPC 通信的方案有诸多缺点，因此新设计的跨核 IPC 系统需要克服这些缺点。但是多个处理器核心之间不共享通用寄存器的问题通过软件是无法解决的，新的设计也只能使用内存来传递 IPC 消息。但多核处理器架构的最大特点是多个处理器核心之间共享 L2 Cache，L2 Cache 用于缓存处理器经常访问的内存，而处理器直接访问 L2 Cache 的速度要比通过前端总线再去访问内存的速度要快得多。本文设计多核间 IPC 机制的出发点就是希望 IPC 消息通过 L2 Cache 来传递，因此使用了邮箱系统。

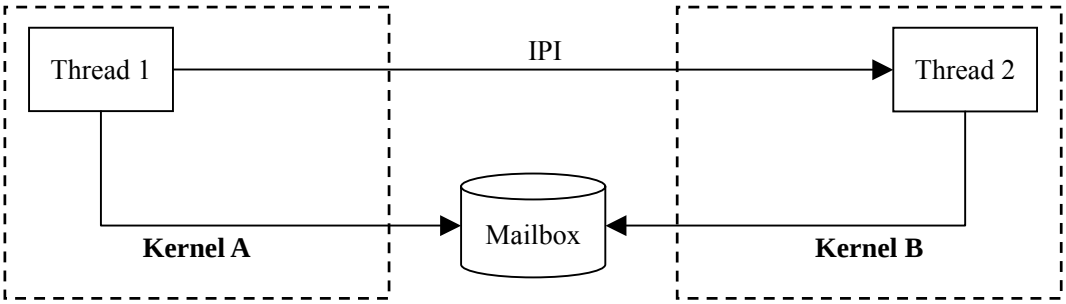

图 4-4 邮箱系统示意图

Fig. 4-4 Mailbox system diagram

邮箱是指内存中的一片地址空间，这片地址空间被所有处理器核心上运行的操作系统内核所共享，每个内核都可以访问和修改邮箱中的数据。邮箱最大的特点是其地址是静态的，在系统启动时就被固定下来，并告知所有的操作系统内核。由于邮箱系统的静态地址被频繁访问，这些地址很可能被硬件放入 L2 Cache，这样对邮箱的访问速度将会大大增加。此外我们还可以通过嵌入汇编给邮箱的访问指令加上 PREFETCHh 前缀，暗示硬件系统把邮箱的地址放入 L2 Cache。

如图 4-4 所示是一个基于邮箱系统的简单的 send IPC。发送线程 Thread 1 发起 IPC 系统调用后，首先把想要发送的消息写到邮箱系统中，然后在寄存器里填上目标接收线程 Thread 2 的线程号（thread id）和 Mailbox 的索引等信息，发送线程 Thread 1 所在的操作系统内核 Kernel A 通过一个 IPI（Inter-Processor Interrupt，处理器间中断）把这些寄存器里的信息发送给接收线程 Thread 2 所在的操作系统能够内核 Kernel B，Kernel B 首先根据中断向量中的线程号找到接收线程，然后根据索引从 Mailbox 中读出 IPC 消息。

使用邮箱系统后也会产生一个新的问题，那就是由于跨核 IPC 和本地 IPC 是两个不同的系统调用，原本的应用程序调用 IPC 时始终还是采用旧的系统调用，无法适用新的发送线程和接收线程在不同处理器核心上的情况。显然我们不可能一一修改所有的应用程序，加上判断条件，让应用程序自己选择使用当前是本地 IPC 还是跨核 IPC，自行选择对应的系统调用。一种可行的方案是提供一个公共的接口，线程调用 IPC 系统调用时实际是调用了这个接口，然后由操作系统判断当前的情况是本地 IPC 还是跨核 IPC，并再调用对应的系统调用。但显然在

这种情况下，对于本地 IPC 也需要这样一次判断和额外的调用，降低了本地 IPC 的效率。如前文所述，本地 IPC 的情况其实要大大多于跨核 IPC 的情况，因此本地 IPC 的效率对微内核操作系统的性能起着关键性的影响，应该采用不会降低本地 IPC 效率的方案。

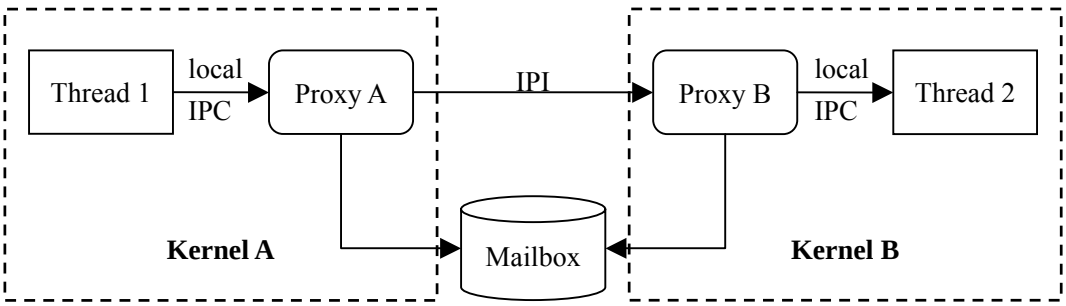

图 4-5 引入代理线程后的邮箱系统示意图

Fig. 4-5 Mailbox system diagram after introducing proxy threads

本设计采用代理线程的方法来实现跨核的 IPC。原本的 L4 本地 IPC 机制保持不变，线程需要调用 IPC 时直接调用原有的本地 IPC 的系统调用接口，同时可以使用基于寄存器的 Short IPC 和基于内存的 Long IPC。如果是一个本地的 IPC，则不需要额外的处理，系统调用完成也即本地的 IPC 完成，流程和原本单处理器的情况一样。如果是一个跨核的 IPC，系统将会使用一个代理线程，先通过本地 IPC 机制把消息发送给代理线程，然后代理线程完成跨核 IPC 的工作。

如图 4-5 所示，每个操作系统内核上都会有一个代理线程来负责跨核的 IPC，图中分别为 Proxy A 线程和 Proxy B 线程。Thread 1 发起一个 IPC 想要发送消息给 Thread 2，但是 Thread 1 和 Tread 2 在不同的处理器核心上，Tread 1 本身不用处理跨核的情况，它依旧调用本地的 IPC 接口来发送消息，操作系统发现消息的接收线程不在当前的内核中时，会把消息转发到 Proxy A，Proxy A 负责跨核的 IPC 通信，它首先把 IPC 消息放到邮箱系统中，然后经过查找发现目标线程 Thread 2 出于 Kernel B 上，于是 Proxy A 发送 IPI 给 Kernel B，让 Kernel B 上的代理线程 Proxy B 从邮箱系统中读取 IPC 消息。Proxy B 读取消息后再通过一个本地的 IPC 发送消息给真正的接收线程 Thread 2，完成这次跨核的线程间通信。

上面的方案很好地提供了透明性，原本的应用程序不用修改就能适用于多核处理器架构，应用程序只需调用本地 IPC，系统会自动地在必要时转换成跨核IPC。但有一个问题是操作系统无法直接知道的，即这次的 IPC 是一个本地的 IPC 还是一个跨核的 IPC。一种解决方案是维护一个查找表，查找表可以用哈希表的数据结构实现来提高查找的效率，保存着每一个线程当前是处于哪个处理器核心上。但是这样的一个查找表需要在所有的内核间同步，其它内核上发生了线程调度也必须在当前内核的查找表上修改；而如果把这个查找放到内核的共享部分上，又涉及需要锁来同步所有内核对查找表的操作，降低了所有内核进行调度的效率。一种改进的方法是查找表只记录当前处理器核心上的线程，如果在查找表里不能找到 IPC 的目标接收线程那就说明接收线程在另一个处理器核心上。但是这样做只是临时解决了问题，当代理线程需要把跨核 IPC 消息发送到接收线程所在处理器核心的代理线程的，还是需要知道接收线程在哪个处理器额核心上，进而知道应给发送跨核 IPC 消息给那个代理线程。除非没一个线程都有一个对应的代理线程，但这种设计的系统开销是不能承受的。

幸运的是，由于本论文的设计采用了分离模型作为微内核操作系统的整体架构，而分离模型本身很好得提供了系统的隔离。本论文的设计在系统启动时，就把所有的线程号分配给了所有的操作系统内核，每个内核会拿到一定范围的连续的线程号，内核拿到的线程号区间不会重叠，也即每个线程号只会分配给某一个处理器核心。此外这个分配也是有规律的，比如最小的线程号首先分配给了第一个处理器核心。因此，操作系统不用维护一张查找表来记录每个线程在哪个处理器核心上，而重要对线程号进行一个除法运算即可。

4.4 本章小结

本章首先介绍了 L4 微内核操作系统 IPC 机制。L4 的 IPC 可以分成 send, receive, wait, reply-wait 和 call 五类，这 5 种 IPC 的功能各不相同。按照存放 IPC 消息的媒介，L4 的 IPC 又可以分成 Short IPC 和 Long IPC。Short IPC 使用寄存器来存放 IPC 消息，速度非常快，但容量有限，如果消息比较长可以用基于内存的 Long IPC，但速度会下降很多。IPC 不仅用来传递消息，还可以用于中断和内

存映射。L4 的 IPC 机制是同步的，发送线程在发起 IPC 后进入等待状态，系统将发送线程加入接收线程的等待着队列，直到接收线程就绪后两边达到握手，IPC 消息完成传递。接着讨论了在 L4 原有的 IPC 机制上进行修改，使之也能应用到发送线程和接收线程在不同处理器核心上的跨核情况的可能性。文章详细分析了这样的一种设计在硬件、同步、时间片捐献机制、透明性和性能方面可能遇到的问题，指出直接在原有的 L4 IPC 机制上修改的方案是不可行的，继而提出使用邮箱系统来处理跨核的情况，发送线程把消息放到固定地址的邮箱系统中，然后通过 IPI 通知对应内核来读取邮箱系统内的消息。这样的邮箱系统设计充分得利用了 L2 Cache,大大提高了跨核的 IPC 性能。为了实现透明性，本论文的设计采用了每个内核上运行一个代理线程的方法。原有的应用程序不用修改，直接调用原有的本地 IPC 接口，操作系统判断是跨核 IPC 时自动转发到代理线程，代理线程负责跨核 IPC 部分。对于判断接收线程所在的处理器核心，本论文的设计利用分离模型总体设计的特点，给每个内核分配了一个线程号的区间，系统可以直接从线程号判断出线程处于哪个处理器核心上。

第5章 系统功能设计

为适应多核处理器架构，除了提供跨核的 IPC 机制，还需要对操作系统的许多方面进行修改。本论文所采用的方案将具体在本章中给与功能设计方案，包括内存管理、线程调度、同步机制和服务应用程序等方面。

5.1 内存管理

5.1.1 L4 的内存管理

在 L4 中，每一个任务对应一个地址空间。地址空间可以有授予（Grant），映射（Map），回收（Flush）三种操作。

授予

拥有地址空间的任务可以把自己地址空间的一部分，以页为单位授予给其它任务。被授予的内存页会从自己当前的地址空间中去除并加入到目标任务的地址空间。任务只能将属于自己地址空间的内存授予其它任务。

映射

任务可以把自己地址空间的一部分，以页为单位映射到其它任务的地址空间。与授予操作不同的是，映射出去的内存页并不会从任务自己的地址空间中去除，而是当前任务和被映射的任务同时可以访问这些内存页。任务只能将属于自己地址空间的内存映射到其它任务的地址空间中。

回收

拥有地址空间的线程可以回收任意通过授予和映射方式允许其他任务访问的内存页。被回收的内存页依旧尅被当前任务访问，但通过直接或间接方式得到这些内存页的其它任务就失去了对这些内存页的访问权限。这种直接回收的方式看似很武断，但一个任务在通过授予或者映射的方式接受来自其它任务的内存页的同时就默认表示愿意接受内存页被回收的可能。

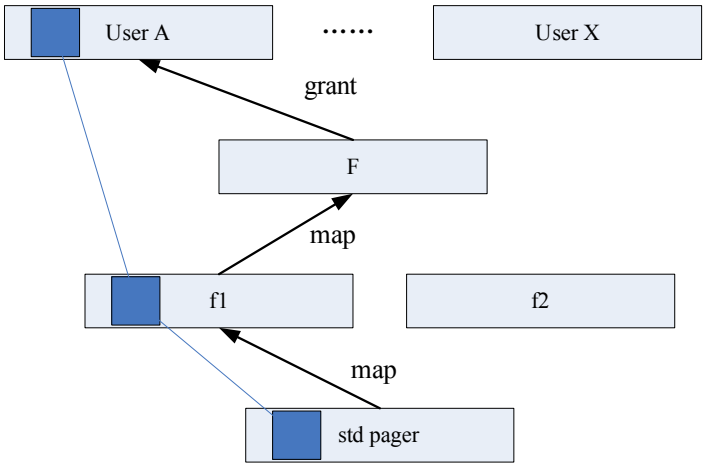

图 5-1 地址空间授予和映射操作示意图

Fig. 5-1 Address space grant and map operation diagram

操作系统将会灵活使用地址空间的这三种操作，特别是在使用一个子系统来负责内存的分配的时候，通过授予而非映射的方式可以大大减轻子系统的负担。如图 5-1 所示，F 是一个 pager，作为底层的两个不同的文件系统 f1 和 f2 的抽象层，负责内存的分配。在图中的例子中，f1 把自己的内存页映射到了 F 中，F 又通过授予的方式把内存页给了 user A。在这种情况下，内存页相当于只被映射到了 f1 和 user A 中，当 std pager 需要收回内存页的时候，只有 f1 和 user A 的地址空间受到影响，而当 f1 需要收回内存页的时候，只有 user A 的地址空间受到影响。如果 F 使用映射而非授予的方式来分配内存的话，它依旧需要去做那些 f1 和 f2 已经在做的工作。

地址空间操作的单位是 fpage，fpage 对应着一片连续的虚拟地址空间。fpage 其实是这片虚拟地址空间对应的内存页表的总和，由于内存页表的大小是 4K 字节，fpage 对应内存区域的大小是 4K 字节的整数倍。

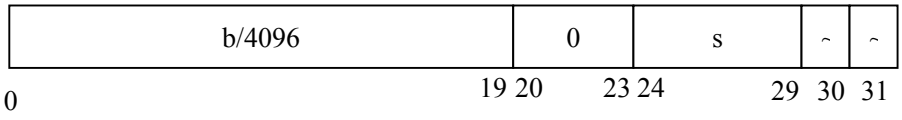

图 5-2 fpage 字各位含义示意图

Fig. 5-2 fpage word bit definition diagram

如图 5-2 所示，描述 fpage 的数据结构是一个 32 位的字，其中 25 位到 30 位表示 fpage 的大小，如果存的数字是 s，fpage 的大小就是 2^s。由于 fpage 的

大小必须是 4K 字节的内存页的整数倍，因此 s 的最小值是 12。前 20 位表示 fpage 对应的虚拟地址的基地值，由于虚拟地址实际上是 32 位表示的，这 20 位其实表示的是 2^s 对齐的地址的前 20 位。因此一个大小为 s，基地址为 b 的 fpage 对应虚拟地址从 b 到 b+2^s-1 的空间。如果 b=0，s=32 就对应了整个 4G 的虚拟地址空间[18][19]。

5.1.2 跨核地址空间共享

对内存管理的设计，其实也就是对地址空间共享程度的设计。按照 L4 的设计，一个任务（相当于一般操作系统中的进程）中所有的线程共享一个地址空间。如果想实现并行计算，即一个任务中的多个线程运行在不同的核上的话，必须让地址空间对所有的内核都是可见的。也就是说更改地址映射等工作必须通知其它核，这无疑增加了系统的复杂性。当然，也可以让每个核上的线程对应一个地址空间，并对所有的地址空间同步，但这无疑成倍地增加了内存的开销，并且同步所有的页表开销也很大。此外设计时还会遇到 L4 中存放内存分配映射关系的数据库 MapDB 是多个核共享还是每个内核拥有一个的问题。由于 MapDB 中保存了内存分配和映射的层次关系，而这种关系难以从一个核复制到另一个核上。除非系统完全不允许线程的迁移，即线程永远都运行在同一个内核上，否则分离的 MapDB 架构难以实现。

综上所述，考虑到多核架构共享 L2 Cache，多核之间通信较快的特点，本设计采用同一任务的多个线程运行在不同核上时共享同一地址空间和 MapDB 的设计。这个架构允许了并行计算，给整个系统带来了很大的灵活性。

此外由于本设计采用了离散的架构设计，并且多个操作系统内核之间共享 MapDB，设计时需要对操作系统内核本身的地址空间做一下处理。对于内核地址空间中非共享的部分，多个内核会分配到不同的物理地址。而对于共享的部分，会使用相同的物理地址。

5.2 线程调度

5.2.1 L4 调度策略

L4 的调度基本单位是线程。对于每个线程都有一个调度上下文（scheduling context）来表征其调度的特性，包括时间片，优先级等等。在任意时刻，只允许一个线程占有 CPU（Fiasco 原本的设计中不支持多核或者多处理器架构，也不支持 Intel 的 Hyper-threading 技术）。线程的当前运行状态保存在内核空间的 TCB（Thread Control Block）中。所以调度线程很大程度上就是交换 TCB。

L4 的调度机制基于静态的优先级，当占用 CPU 的线程执行完毕的时候，系统在就绪队列中找到优先级最高的线程占用 CPU。当同时有多个线程处于最高的优先级上时，系统会调度这些线程轮流占用 CPU 执行，平分执行时间。

系统保存一个就绪队列（ready-list）来决定下面应该运行哪个线程。就绪队列保存了所有处于就绪状态的线程。系统在当前的线程执行结束后找到就绪队列的第一个线程，调入 CPU 开始执行。

就绪队列通常能给操作系统的调度带来便利，但是在有些情况下，维护就绪队列会变得非常麻烦。比如某个线程要进行一个 IPC 调用，发送线程发送一个消息给接收线程，然后等待接收线程发送返回消息。这个过程分为以下 4 个步骤：

1. 发送线程线程发出消息，离开就绪队列进入等待状态。

2.接收线程进入就绪状态，被加入就绪队列，操作系统完成消息的发送。

3. 接收线程发送返回的消息，自己离开就绪队列进入等待状态。

4. 发送线程重新变成就绪状态并被加入就绪队列，接收来自接收线程的消息。

在这个过程中的每一步都要修改就绪队列，这大大降低了 IPC 的效率，而 IPC 的效率是影响整个微内核操作系统的关键。对于这种情况，L4 提出了 lazy-scheduling 的策略，一方面对于进入等待状态的线程，依旧会被保留在就绪队列中，但操作系统在寻找就绪线程时会跳过等待状态的线程；另一方面，对于执行状态的队列，不需要保存在就绪队列中。这样就避免了 IPC 过程中所有的对就绪队列的修改，大大提高了效率。当然系统也提供了 deceit-bit 这样一个标志

位来屏蔽 lazy-scheduling 调度策略，此时等待状态的线程会被移出就绪队列并且 IPC 中需要被执行的线程会被先加入就绪队列中，这在某些特殊应用下提供了灵活性。

此外系统也用到了时间片捐献的方法来提高调度效率。对于 IPC 的情况，当发送线程完成发送，需要调度到接收线程的时候，系统并不进行调度上下文的切换，而是直接在发送线程的调度上下文中运行接收线程。这样接收线程就可以先继续使用发送线程的时间片，这样接收线程可以更快得完成工作并返回到发送线程，发送线程也会从这样的时间片捐献中受益。

如大多数使用静态优先级的操作系统一样，L4 也会遇到优先级反转（priority inversion)的问题。比如有如下场景：当一个高优先级的线程向一个低优先级的线程请求服务的过程中，高优先级线程进入等待状态，低优先级线程处理服务请求，而此时系统又产生一个中优先级的线程，由于它的优先级比低优先级的服务线程高，这个中优先级线程会占有 CPU，间接阻塞了优先级比它还高的高优先级线程。对于优先级反转的问题，各个操作系统会采用优先级置顶或者优先级继承的方式。优先级置顶是指服务进程在处理服务请求时会临时达到最高的优先级，保证自己能占有 CPU 并完成高优先级线程的服务请求，而优先级继承是指服务线程临时继承了请求服务的高优先线程的优先级，并处在这个优先级上处理请求，这就保证了其它中优先级的线程无法抢占 CPU 资源。而 L4 采用了它的时间片捐献方式来处理优先级反转的问题。当高优先级的线程请求服务线程服务而自己进入等待状态时，他把自己的时间片捐献给服务线程，让服务线程用自己的时间片来处理自己的服务请求，此时服务线程也就运行在自己的优先级上，不会被中优先级的线程抢占 CPU 资源。

5.2.2 核间调度

对于线程调度，最理想的情况自然是完全自由的全局调度，内核把线程分配到空闲的核上，并进行全局的负载平衡。但全局的调度有诸多问题。首先，L4 的调度策略是最高优先级的线程占有 CPU，如果有多个线程有相同的优先级，则轮流被执行。如果在多核架构下，这些最高优先级的线程数少于处理器核心的数量，则就算有核空闲也只能空跑，造成了资源的浪费。其次，L4 有所谓“捐

赠”的机制，被阻塞的线程可以把自己的时间片贡献给依赖的线程，使自己需求的资源尽快释放或者自己想要的服务尽快被执行[20]。但对于多核架构，线程所拥有的时间片只是针对自己所在的核心，如果要把依赖的线程迁移到自己所在的核运行自己的时间片，然后再迁移回去，情况就变得很复杂。此外，由于本设计使用了离散模型，同步全局的等待队列也是件开销很大的事情。

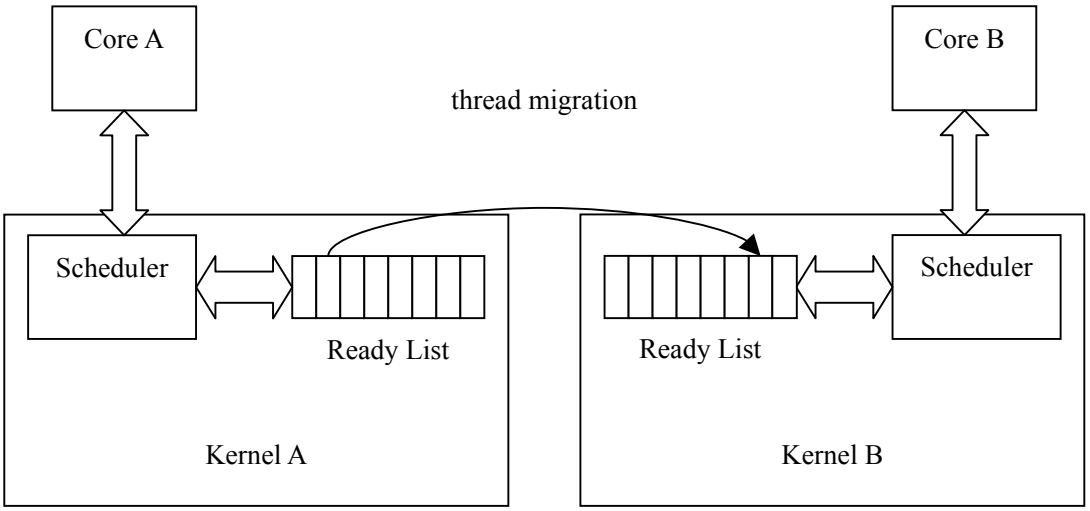

图 5-3 跨核调度机制示意图

Fig. 5-3 Cross-core schedule mechanism diagram

因此本设计对于细粒度的调度只支持在操作系统内核内部，即同一个处理器核心中进行，系统也支持同一个核内部的时间片捐赠。如图 5-3 所示，系统会提供一种称为线程迁移的方法，把线程从一个核转移到另一个核。具体操作就是保存线程的状态并在当前的核上删除，然后在另一个核上重新创建线程。对于多核之间的负载平衡，系统通过用户态的应用程序通过线程迁移实现。

5.3 锁和同步机制

5.3.1 L4 的同步机制

L4 提供两种机制来进行同步操作，分别为 CPU-lock 和 switch-lock。

CPU-lock 是通过关中断的方式来实现的。当一个线程访问共享的资源时，如果被另一个线程中断，另一个线程也访问了该内存，并修改了内存中的数据，当中断返回到原来的线程继续访问那段内存时，数据已经和中断前不同了。为了避免这种情况的发生，最简单的方法是线程在访问共享资源时屏蔽所有中断，这样就能保证它对共享资源的访问是原子操作。在 L4 中是通过 CPU-lock 来实现关中断地访问共享资源。这种方式适用于那些频繁发生并且短时间内可以完成的同步操作，对于需要长时间才能完成的同步，如果一直屏蔽中断，可能造成重要的外部控制信号丢失。

switch-lock 是通过锁的机制来实现同步的。每个共享资源都对应于一个锁，当线程想要访问该共享资源是，首先需要获得该资源的锁，此时当其它线程想要访问该资源时就会无法获得锁进而无法访问该资源，进入阻塞状态。知道拥有锁的线程完成了对资源的访问并释放了锁，其它线程才能再次获得锁并访问资源。但是一般的锁的效率并不高，L4 提出了一种称为时间片捐赠的方式来提高锁的效率。假设线程 A 首先获得了资源的锁，此时线程 B 也想访问该资源，但无法获得锁，进入阻塞状态，一般情况下线程 B 资源会处于一个循环中不断查询锁的状态，直到锁被线程 A 释放。但 L4 允许线程 B 吧自己的时间片捐献给线程 A（当然根据 L4 的调度策略，需要线程 B 的优先级大于等于线程 A 的优先级才有效）。线程 A 可以使用线程 B 的时间片来对共享资源进行访问，线程 A 拥有更多的时间片来访问共享资源意味着更早得完成访问并释放锁，线程 B 也就可以更早得得到锁来访问资源，这当然对线程 B 也是有利的。这里的锁的时间片捐献机制和 IPC 中发送线程和接收线程之间的时间片捐赠有相似之处。switch-lock 同步机制比较适用于需要同步的时间较长并不会频繁发生的情况。

5.3.2 多核间同步

L4 中提供的关中断和 Helping Lock 两种机制都难以适应多核架构的环境。关中断只能屏蔽来自当前处理器核心的中断，而 Helping Lock 是指被阻塞的线程贡献时间片给拥有锁的线程来加快资源释放，这在多核架构下需要线程在两个核间来回迁移，实现机制复杂。综上所述，关中断和 Helping Lock 依旧会被保留，但只能用在处理器核心内部的同步，还需要寻找其它方法来实现多核之间的锁和同步。

Intel 架构的多核处理器提供一种叫总线锁的机制。系统可以使用 LOCK#信号或者 LOCK 指令前缀来在执行访问内存命令时锁定总线，其它处理器核心对总线的访问会被屏蔽。但正如 Intel 的开发手册所述，总线锁的作用是保证对内存的 byte 和 word 级别的读写是原子操作，如果访问的是大块的内存区域，总线锁有可能被系统中断。

对于多核间的同步，本设计采用自旋锁或者处理器间中断的方式实现。自旋锁是最简单的同步机制，线程在等待循环中不断查看自旋锁的状态，直到锁被释放。对于时间较短并且发生频率较低的同步，系统可以采用自旋锁的方式。Intel 64 以及 IA-32 架构提供 IPI 机制（Inter-processor interrupts，处理器间中断）。一个核可以通过 IPI 使另一个核中断。操作系统可以通过发送 IPI 给目标核，使之执行中断向量中指定的线程。

基于多核架构的特点，本设计使用前文所述的邮箱系统来进行同步。所谓邮箱系统就是内存中一些所有内核都知道的固定地址，一个核将需要同步信息放到邮箱系统中并通过 IPI 来通知其它核来读取。由于邮箱的内存地址会被频繁使用，这些内存地址很可能被放到 L2 Cache 中，而又因为多核架构共享 L2 Cache，对邮箱的访问速度会很快。此外还可以在实现邮箱时，通过 PREFETCHh 指令来暗示处理器核心把邮箱对应的内存读入 L2 Cache，更加保证了邮箱的读写速度。

5.4 服务应用程序

微内核操作系统的内核只提供了一些基本的机制，具体的操作系统服务通过服务应用程序的方式来实现。而对于多核处理器架构，本论文又采用了分离模型作为总体设计，就可能出现如下的情况：请求服务的客户线程和服务应用程序线程不在同一个处理器核心上，也即不在同一个操作系统内核上。本文提供了三种模式来处理这些情况，分别为独立模式、主从模式和分布式模式。

5.4.1 独立模式

独立模式是最简单的一个模式。它是指在每个操作系统内核上运行一个独立的服务程序，每个服务程序只需处理来自自己所在的内核的其他应用程序的服务请求，不会处理甚至不会接收来自其它内核的服务请求，因此各个操作系统内核上的服务程序之间也不需要任何通信和同步。

在独立模式中，服务程序和客户应用程序之间通过本地 IPC 通信，并通过操作系统内核提供的系统调用来实现各种操作系统服务，这种模式和原本 L4 上的模式完全相同，因此有很好的兼容性。原本 L4 上的应用程序可以不经修改，直接以独立模式在本论文设计的新的支持多核架构的操作系统上运行。

独立模式是最简单，也是效率最高的一个模式，如图 5-4 所示。它适用于那些作用范围限制在操作系统内部或者同一个处理器核心的服务应用程序，比如分配操作系统资源的服务程序就很适合用独立模式实现。另外对于发送服务请求的客户线程可能分布在不同操作系统内核上的情况，如果这些线程互相之间独立，不需要相互作用，而服务的要求的相应时间又比较短，可以在多个内核中各起一个独立的服务应用程序来提供服务。

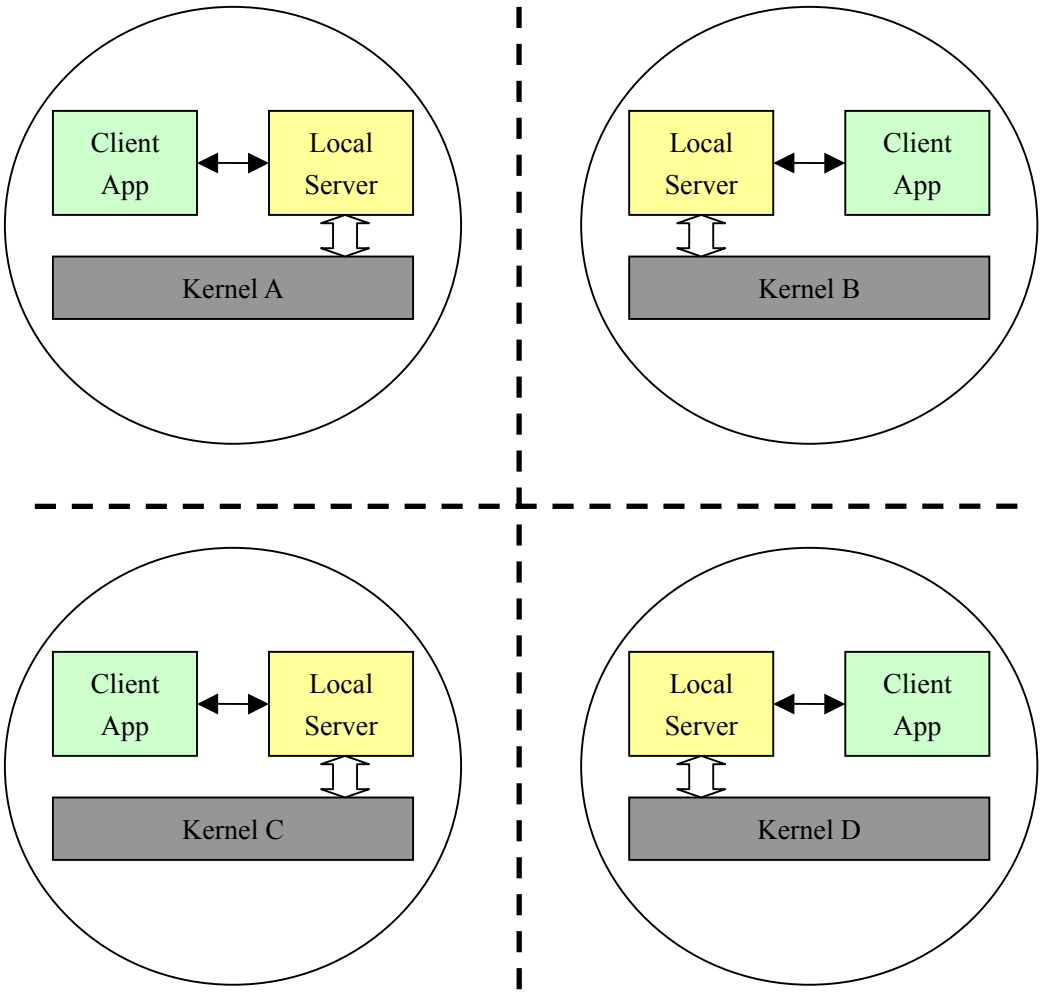

图 5-4 独立模式示意图

Fig. 5-4 Independent pattern diagram

5.4.2 主从模式

主从模式中有一个主服务程序，运行在某一个操作系统内核上，负责使用内核提供的系统调用处理所有的服务请求。而在其它所有的内核上会运行一个从服务器，从服务器不会调用系统调用来处理请求，但它可以接收来自本内核上的客户应用程序发送的请求，并把请求通过跨核 IPC 发送给主服务程序，主服务程序完成服务处理后再把结果通过跨核 IPC 发送给了从服务器，从服务器再通过本地 IPC 发送给客户应用程序。

主从模式可以适用于客户应用程序在不同处理器核心上的情况。如图 5-5 所示。特别是当 L4 原有的服务应用程序如果需要应用在这种情况时，很适合于

使用主从模式。因为主从模式只需要实现一个很简单的从服务程序来接收来自其它内核的请求，再对原本的服务程序进行简单修改，增加一个从服务程序接收请求并返回结果给从服务程序的接口即可，可以最大限度得保留原有的代码。

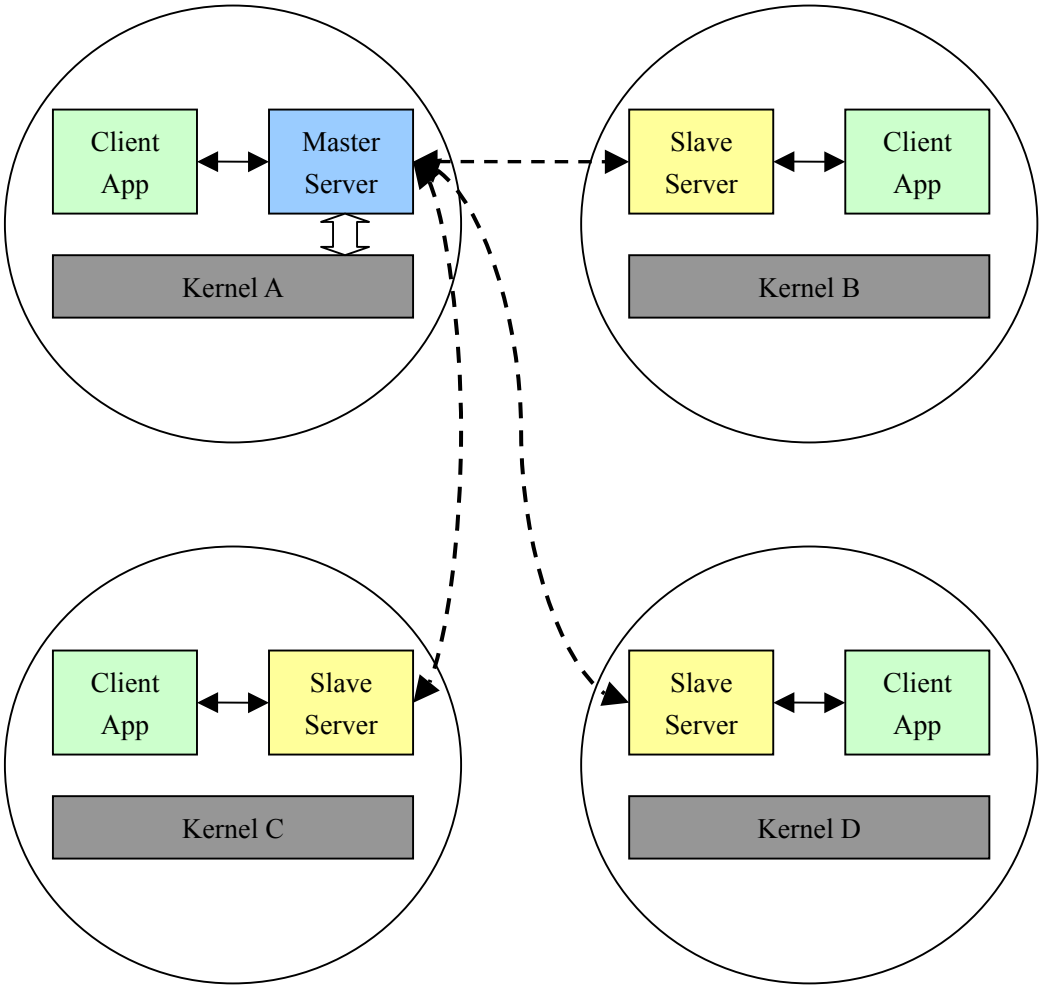

图 5-5 主从模式示意图

Fig. 5-5 Master-slave pattern diagram

5.4.3 分布式模式

主从模式虽然能处理发送服务请求的客户应用程序分处不同的操作系统内核上的情况，但是对于从服务程序所在的内核上的客户程序，每次请求都要通过跨核 IPC 发送到主服务程序那边处理再返回，效率不会太高。因此本论文还提供分布式模式，如图 5-6 所示。

分布式模式在每个内核上都有一个分布式的服务程序，所有的分布式服务程

序的关系是平等的，每个服务程序都有完整的功能来处理来自自己所在内核的请求。分布式模式和独立模式最大的区别是所有的分布式服务程序其实是一个整体，系统提供跨核的数据结构用于所有分布服务程序的同步。因此分布模式提供了比主从模式更高的效率，但也需要额外的开发工作来大量修改原有的 L4 服务应用程序的代码来加入新的同步机制。

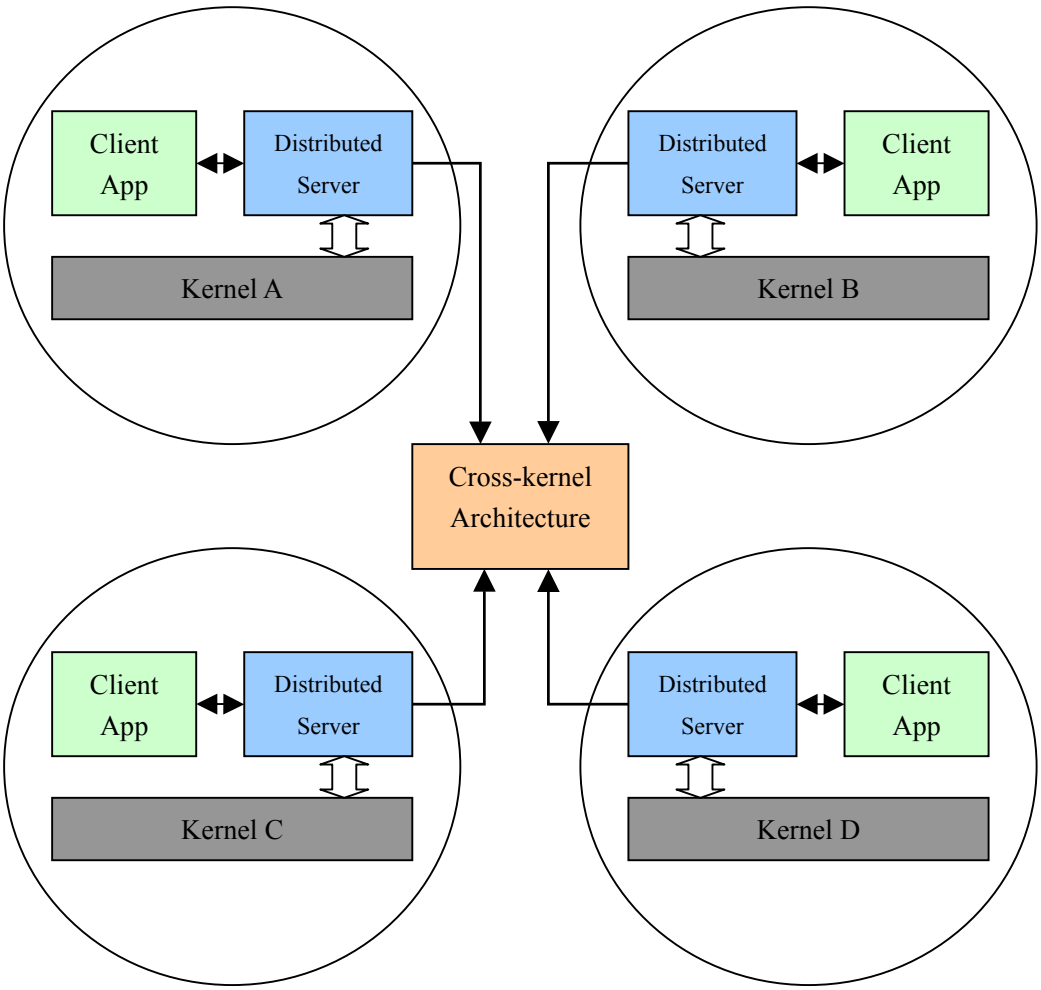

图 5-6 分布式模式示意图

Fig. 5-6 Distributed pattern diagram

5.5 本章小结

设计一个应用于多核处理器架构的微内核操作系统，除了线程间通信之外，还需要考虑内存管理、线程调度、同步机制和服务应用程序方面的设计，本章一一分析了 L4 在这些方面的机制，并提出了为了适应多核处理器架构而采用的设计方案。

在内存管理方面，本设计允许同一个任务的多个线程同时运行在多个处理器内核上，也即地址空间的覆盖域可以跨核，到保存地址空间映射关系的 MapDB 只有一个实例并被所有的操作系统内核共享。在线程调度机制上，本设计允许处理器内核内部的细粒度调度，并提供线程迁移的机制来实现多核间的调度。在所和同步机制方面，本设计保留原有的 cpu lock 和 swich lock 作为本地同步机制，而使用自旋锁和 IPI 来实现跨核的同步。对于微内核之上运行的服务应用程序，本设计提供独立模式、主从模式和分布式模式三种选择。

第6章 系统功能实现

6.1 环境、设备及方法

在软件方面，本设计最终选用了德国 Dresden 科技大学开发的 Fiasco 作为研究基础，Fiasco 作为 L4 规范的最流行的一个实现，功能完善并且版本稳定，非常适合于学术研究。而 L4 作为第二代微内核操作系统的代表，不仅具有微内核操作系统的诸多优点，而且经过对 IPC 的特别改善，性能方面也十分出色。在开发工具上，使用 Linux 作为开发平台，并使用 vi+doxygen 作为开发工具，gcc 作为编译工具，svn 作为版本控制工具。

在硬件平台上，在硬件平台上，本设计在初期使用 VMWare 虚拟机调试。由于开发涉及底层部分，使用真机不断重启效率很低，而 VMWare 不仅能完整得模拟出双核处理器的硬件架构，而且启动很快，能大大提高调试速度，而且 VMWare 提供的录像功能非常适用于调试微内核操作系统的启动部分。而在测试阶段，为了在实验硬件平台上测试微内核操作系统的性能，本设计使用了基于 Intel Core2 Duo Q965 芯片组的开发平台，CPU 为 Core2 Duo 2.13GHz。

6.2 邮箱系统功能实现

为了实现邮箱系统，我们需要修改 L4 原本的 IPC 机制，使之可以适应多核架构处理器的情况。系统将会根据接收线程的线程号来判断这次 IPC 是一次内核本地的 IPC 还是跨核的 IPC，如果是跨核的 IPC，系统将会把 IPC 消息发送给当前内核的代理线程，代理线程会把 IPC 消息放到邮箱系统中，并通过 IPI 来通知对应的内核去读取，对应内核上的代理线程从邮箱读取消息后再通过本地 IPC 发送给真正的接收线程。

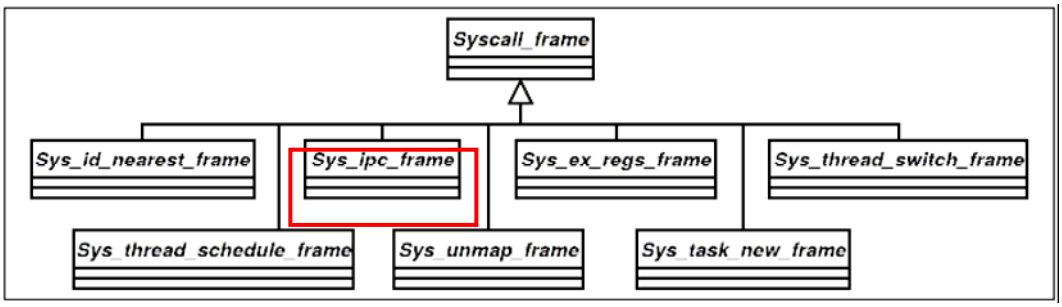

图 6-1 系统调用类图

Fig. 6-1 System call class diagram

如图 6-1 所示，Syscall_frame 类定义了系统调用中用到的寄存器。而所有的五个系统调用都有自己的类继承 Syscall_frame 类，在子类中各自定义了在自己的系统调用中各个寄存器所代表的含义。IPC 系统调用对应的是 Sys_ipc_frame 类，具体寄存器含义如图 6-2 所示。

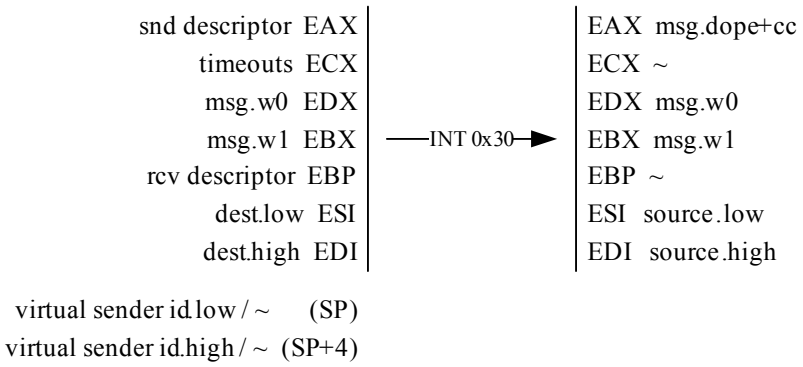

图 6-2 IPC 系统调寄存器含义示意图

Fig. 6-2 IPC system call register meaning diagram

snd descriptor

EAX 寄存器中存放的 snd descriptor 表示了 IPC 发送阶段的属性，如果 snd descriptor 前 30 位全位 0，则为 short IPC，消息存放在寄存器中，否则为 long IPC，消息存放在前 30 位所对应的内存中。此外 snd descriptor 还有标志位表示此次 IPC 是否是一个内存页映射操作。

rcv descriptor

和 snd descriptor，定义了 IPC 接收阶段的属性。rcv descriptor 有标志位定义是只接收来自 dest id 线程的消息还是接受任意线程的消息。

dest low, dest high

如图 6-3 所示，线程号有 64 位，dest id 所以分别存放在两个寄存器中。系统内核将会查看 dest id 是否落在自己所拥有的线程号范围内，如果是则为本地 IPC；如果 dest id 不在当前内核拥有的线程号范围内，则说明是一个跨核的 IPC，系统通过对 dest id 换算可知道目标线程在那个内核中。系统把 dest id 存放到 IPC 消息的最前面，并把对应寄存器的 dest id 换成当前内核的代理线程，进行跨核 IPC 通信。

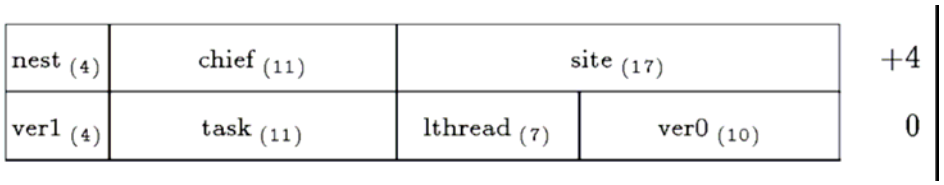

图 6-3 线程号说明图

Fig. 6-3 Thread id definition diagram

source.low, source.high

发送线程的线程号。如果这次 IPC 实际上是一次中断，则 source id 为中断号。

msg.w0 msg.w1

用于存放 IPC 消息的寄存器，即使是 Long IPC，前两个字的消息也会被放在这两个寄存器中。如果是在发送阶段，寄存器中存放的是将要发送的消息，而在接收阶段，寄存器存放的是接收到的消息。此外还有一个特别的地方是，在发送阶段，如果是 Long IPC，内存和寄存器中都会有消息的前两个字的内容，而在接收阶段，前两个字只会出现在寄存器中，内存中的消息从第 3 个字开始。

msg.dope + cc

msg.dope 部分描述了接收到的消息的属性。而 cc 部分则是一系列的标志位描述了此次 IPC 是否成功，如果失败了是什么原因造成的。

timeouts

描述了发送和接收阶段线程等待多久之后没有得到响应视为此次 IPC 失败。

6.3 邮箱系统性能分析

在 Intel IA32 架构中，对于内存缓存同步使用 MESI 协议。内存共享协议的目的是使得多个处理器或者多核上不共享的 Cache 的内容保持统一。MESI 协议名称中的每个字母分别代表了缓存中的缓存行所能处于的状态。每个缓存行都有一个两位的标志来来表示自己当前属于四个状态中的哪一个，而四个状态分别为：

修改(Modified)：此时缓存行缓存了来自内存的信息后又被处理器所修改。

独占（Exclusive）：此时缓存行缓存了来自内存的信息后，没有其他处理器或处理器核心缓存了同样的内存地址，此缓存行内容也未必修改过。

共享（Shared）：此时至少有两个处理器或者处理器核心的缓存行缓存了来自同样内存地址的信息，并且此缓存行内容也未被修改过。

无效（Invalid）：此时缓存行中的内容无效。可能是还没有内存被缓存或者此缓存对应的内存地址被另一个处理器或处理器核心所修改。

对于多处理器架构，每个处理器有各自的 L1 Cache 和 L2 Cache，MESI 的四种状态用于多处理器之间的同步。而对于多核架构，所有的处理器核心共享 L2 Cache，但有各自的 L1 Cache。因此 L1 Cache 的 MESI 状态对应了处理器核心之间的缓存同步状态，而 L2 Cache 的 MESI 状态对应的是有多个多核处理器时，多个处理器之间的 L2 Cache 同步，而这种情况不在本文的讨论范围内。因此对于本文所讨论的多核架构，L2 Cache 不需要同步，也不会处于共享状态，并且无效状态只可能是还没有内存加载如缓存。

对于加载入缓存的邮箱系统的访问，系统底层会有如下几个步骤：

1. 首先是一个处理器核心访问了邮箱系统中的某个内存地址，内存被加载入 L2 Cache，随后继续加载入 L1 Cache 并且对应缓存行的标志位成为独占。
2. 此后可能发生的步骤是另一个处理器核心也访问了邮箱系统，此时两个处理器核心的对应缓存行的状态都会被设成共享,这是一个 e2s 过程(独占到共享，Exclusive to Shared，后文类似)。
3. 处理器核心对缓存的内容进行了修改，当缓存行处于独占状态时，会有一个 e2m 过程，缓存成为修改状态。而当缓存处于共享状态时，对于发

起修改的处理器核心的缓存是一个 s2m 过程，而对于另一个处理器核心的缓存则是一个 s2i 的过程，对应缓存行处于无效状态。

4. 当另一个处理器核心需要读取邮箱系统的内容，首先对应 L1 Cache 缓存行的内容被写道 L2 Cache，然后另一个处理器核心的缓存通过 i2s 过程，从 L2 Cache 读取内容加载到 L1 Cache 以供访问。此时两个处理器核心的缓存的对应缓存行都处于共享状态。

表 6-1 缓存访问速度比较

Table 6-1 Comparison between cache access speed

案例	数据存放位置	延时
L1 to L1 Cache	L1 Cache	14 core cycles + 5.5 bus cycles
Through L2 Cache	L2 Cache	14 core cycles
Through Memory	Main Memory	14 core cycles + 5.5 bus cycles + ~40-80 nsec depending on FSB and DDR freq.

如表 6-1 所示，根据 Intel 提供的数据[21]，当访问的数据处于 L2 Cache 中，整个同步周期只需 14 个内核时钟周期，而当数据处于内存中时，则需要 14 个内核时钟周期加上 5.5 个总线时钟周期再加上 40 到 80 纳秒的总线延时（具体延时取决于不同型号的前端总线和 DDR 内存总线频率）。而对于多处理器架构，其 L2 Cache 也需要在多处理器间同步，也要经过 i2e, e2s, s2m, s2e, m2s, i2s 过程，这也意味着还需要额外的时间来完成整个数据同步操作。由此可见，本设计的邮箱系统在邮箱数据处于 L2 Cache 中时可以大大提高访问速度，进而提高跨核 IPC 的速度乃至整个系统的效率。

6.4 系统性能测试

x86 汇编指令在系统中的执行时间如表 6-2 所示。数据测试方法是连续执行对应指令 10000 次，并使用 RDTSC 指令统计执行前后的时间戳，每条指令的平均执行时间。这些数据一方面可以表征本文的 Core2 Duo 2.13GHz CPU Q965 芯

片组实验平台的基本性能；另一方面也给出了操作系统和邮箱系统需要用到的关键指令的执行时间。

表 6-2 关键 x86 汇编指令执行时间表

Table 6-2 Critical x86 instruction execution time table

Instruction	Time (cycles)
1: dec ECX, jnz 1b	1.0
wbinvd	409475.2
invlpg	242.0
read CR3	16.0
reload CR3	175.0
clts	52.0
cli + sti	41.1
set CR0.ts	181.0
in8 (PIC)	1987.5
int8 (iodelay)	1912.6
out8 (PIC)	1974.7
get ES	1.0
set ES	16.5
modify ES + load ES:mem	24.7
int + iret	483.0
sysenter + sysexit	142.0
sysenter + iret	318.9
push EAX + pop EAX	6.0
push Flags + pop EAX	20.0
rdtsc	65.0
rdpmc(0)	54.0
load data TLB (4k）	14.8
load code TLB (4k)	47.9

如图 6-4 所示为基于寄存器的 IPC 的执行时间分布图，基于寄存器的 IPC，也即 Short IPC 是指 IPC 的发送端和接收端都通过寄存器来存放消息，其执行时间分布主要为 128 cycles, 152 cycles，160 cycles 这三个值，此外也有少量大于 320 cycles 的情况。而对于使用内存来存放消息的 Long IPC 的情况，执行时间并不是某几个确定的值，而是分布在 1200 ～ 1600 cycles 和 2800 ~ 3200 cycles 这两个区间中，这应该主要和是否产生缺页异常有关。而对于地址空间页映射的情况，执行时间分布在 8000～8500 cycles 左右，比较集中。

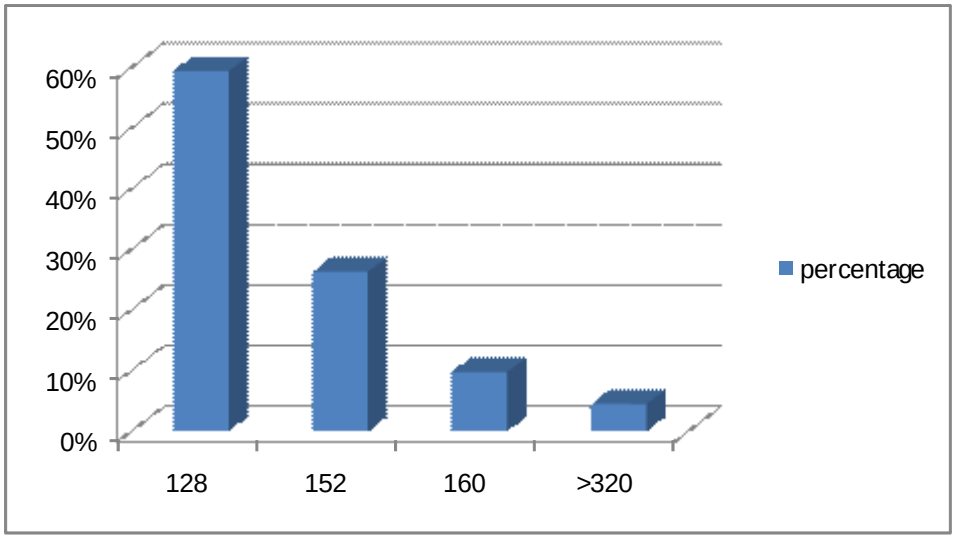

图 6-4 基于寄存器 IPC 的执行时间分布图

Fig. 6-4 Register based IPC execution time distribute diagram

表 6-3 IPC 执行时间表

Table 6-3 IPC execution time table

Type	Ave Original Fiasco Perf	Ave Core-local Perf	Ave Cross-core Perf (Expected)
short IPC (register to register)	142.3 cycles	147.5 cycles	3834.5 ~ 5981.3 cycles
4 word long IPC (memory to memory)	2372.0 cycles	2386.3 cycles	8312.1 ~ 10458.9 cycles
flexpage mapping	8236.8 cycles	8250.4 cycles	14176.2 ~ 16323.0 cycles

由于对于邮箱系统的性能测试需要整个系统都实现后才能进行。然而本文相关的课题时间跨度较长，工作量上也无法让一个人同时承担所有的设计和实现工作，实验室下一届的同学将会继续接手课题，因此本文主要给出了处理器核心本地 IPC 的性能数据，并通过本地 IPC 的数据结合 MESI 相关的性能分析来推算跨核 IPC 的性能数据。

如表 6-3 所示，对于 IPC 发送端和接收端在同一个处理器核心上的本地 IPC 情况，测试结果表明，基于寄存器的 Short IPC 平均需要 147.5 cycles 的执行时间；而对于使用内存来存放消息的 long IPC 则需要平均 2386.3 cycles 的执行时间；对于地址空间页映射的情况，平均执行时间为 8250.4 cycles。

这项性能对比原来未被修改的 Fiasco 在本平台上的实验数据：short IPC

142.3 cycles，long IPC 2372.0 cycles，地址空间页映射 8236.8 cycles。本系统的本地 IPC 性能几乎没有下降，这是因为对于本地 IPC，系统基本保留了原有的 IPC 设计以实现透明调用，只是增加了一个判断接收端是否在当前内核上的步骤，并且使用了分支预测技术默认是本地的 IPC，因此对最为频繁发生的本地 IPC，性能基本上没有受到影响。

而对于 IPC 发送端和接收端在不同处理器核心上的跨核 IPC 情况，对于 Short IPC，整个跨核 IPC 的过程包括两个本地的 Short IPC 用于和代理线程通信，一个基于邮箱系统的跨核消息传递，以及一个 3300 cycles[22]左右的 IPI 来进行通知，其中邮箱系统步骤可以通过共享的 L2 Cache 的提高访问速度，根据邮箱内容是否被调入 L2 Cache，总的执行时间是 8312.1 ~ 10458.9 cycles 的浮动值。对于 Long IPC，为了实现透明调用，跨核 IPC 首先需要两个本地的 IPC 和代理线程间的通信，然后需要一个 3300 cycles 左右的 IPI 来进行通知，并通过邮箱系统发送跨核的消息，总的执行时间是 8312.1 ~ 10458.9 cycles 的浮动值。而对于地址空间页映射的情况，也需要通过内存来存放消息，因此可以去除页表分配等时间，将消息发送时间以类似 Long IPC 的情况换算，最后得到执行时间为 14176.2 ~ 16323.0 cycles。

6.5 本章小结

本章介绍了具体的系统功能实现。首先介绍了本文采用的软硬件平台，软件是在 L4 规范的 Fiasco 实现开源代码的基础上修改，硬件先后采用 VMWare、Q965 和 MID。本章随后介绍了 IPC 系统调用寄存器含义等邮箱系统实现细节，并对邮箱系统的性能进行了分析。最后给出了在实验平台上的性能测试数据，主要是处理器核心本地 IPC 的性能数据，并通过本地 IPC 的数据推算了跨核 IPC 的性能数据。

第7章 总结与展望

7.1 主要结论

本文主要设计了一个基于多核处理器架构的微内核操作系统。微内核架构的内核部分较小，操作系统所需的服务并不全部实现在内核内，而是在内核仅提供实现的基本机制，而把服务的实现放到了用户态的应用程序中。其在可靠性、实时性和安全性方面的优势使之非常适用于嵌入式环境。然而随着多核处理器架构在嵌入式领域普及，微内核操作系统却不能支持多核处理器硬件架构，这正是本论文的出发点。事实上，已经有一些国外的学者在为 L4 微内核操作系统增加对多处理器架构方面的研究进展。虽说这些文献对本论文很有参考价值，但由于多处理器架构和多核架构在是否共享 L2 Cache 这点上的巨大区别，还是很有必要针对多核架构共享 L2 Cache 的特点设计效率更高的微内核操作系统。

本文针对微内核操作系统和嵌入式环境应用的特点，使用离散模型作为整体设计的方案，所谓离散模型是在每个处理器内核上运行一个操作系统微内核，多个操作系统内核之间通过一定机制同步。这样的设计在嵌入式环境中会有很多的应用，比如让一个内核作为另一个内核的热备份来提高安全性；又如两个核分别调度实时和非实时的应用程序来实现很好的隔离；还可以用一个内核上网购物，而另一个内核用于现金结算来实现很好的安全性。

微内核操作系统相比单内核操作系统有很多优点，但最大的缺陷是性能问题。由于微内核操作系统把具体的操作系统服务放到了用户态的服务应用程序，每次使用这些服务都需要通过 IPC 来发送请求给这些服务程序，因此 IPC 的效率对于微内核操作系统至关重要。本文着重介绍了对于多核处理器架构下的跨核 IPC 的设计。为了充分利用多核处理器架构共享 L2 Cache 的特点，需要跨核 IPC 传递消息使用的内存地址是固定的，这样才有更大的机会被调入 L2 Cache。因此本文使用邮箱系统来处理跨核的情况，发送线程把消息放到固定地址的邮箱系统中，然后通过 IPI 通知对应内核来读取邮箱系统内的消息。由于邮箱的地址固定，因而充分得利用了 L2 Cache,大大提高了跨核的 IPC 性能。为了实现透明性，

本论文的设计采用了每个内核上运行一个代理线程的方法。原有的应用程序不用修改，直接调用原有的本地 IPC 接口，操作系统判断是跨核 IPC 时自动转发到代理线程，代理线程负责跨核 IPC 部分。对于判断接收线程所在的处理器核心，本论文的设计利用分离模型总体设计的特点，给每个内核分配了一个线程号的区间，系统可以直接从线程号判断出线程处于哪个处理器核心上。

设计一个应用于多核处理器架构的微内核操作系统，除了线程间通信之外，还需要考虑内存管理、线程调度、同步机制和服务应用程序方面的设计，本文一一分析了 L4 在这些方面的机制，并提出了为了适应多核处理器架构而采用的设计方案。在内存管理方面，本设计允许同一个任务的多个线程同时运行在多个处理器内核上，也即地址空间的覆盖域可以跨核，而保存地址空间映射关系的 MapDB 只有一个实例并被所有的操作系统内核共享。在线程调度机制上，本设计允许处理器内核内部的细粒度调度，并提供线程迁移的机制来实现多核间的调度。在所和同步机制方面，本设计保留原有的 cpu lock 和 swich lock 作为本地同步机制，而使用自旋锁和 IPI 来实现跨核的同步。对于微内核之上运行的服务应用程序，本设计提供独立模式、主从模式和分布式模式三种选择。

7.2 研究展望

多核处理器架构未来的发展有两个趋势。一个趋势是海量的内核，未来的处理器中可能有 64 个，1024 个甚至更多的处理器核心。显然在这样的架构下，依旧使用离散模型作为微内核操作系统的总体架构，在每个处理器内核上运行一个微内核不是一个理想的选择。我们可以把离散模型和全局模型整合成一个混合模型，在少量的处理器核心上运行微内核，而把剩下的处理器核心作为处理器资源分配给每一个操作系统微内核，微内核内部有调度机制来决定到底把线程调度到哪个属于自己的处理器核心上运行。

多核处理器架构的另一个趋势是非对称架构。所谓非对称架构是指处理器核心被分成两类：一类负责管理其他处理器核心，另一个负责执行具体的程序。比如 Play Station 3 上使用的 Cell 处理器就有 1 个 PPE（Power processing element，功率处理单元）和 8 个 SPE（Synergistic Processing Elements，协处理单元）组成。

对于这样的非对称多核处理器架构，也可以使用离散模型结合全局模型的混合模型来应对，在主处理单元上运行操作系统微内核，而把协处理单元作为计算资源调度。

综上所述，本文下一步可能的研究方向可以是针对非对称海量内核架构硬件平台的混合模型的微内核操作系统设计。

参考文献

[1] Marcus Völp. Prototypical Design and Implementation of L4-SMP Microkernel Mechanisms [Dissertation]. Karlsruhe: TU Karlsruhe, 2002.

[2] Sven Schneider. Multiprocessor Support for the Fiasco Microkernel [Dissertation]. Chemnitz: Technical University of Chemnitz, 2006

[3] Volkmar Uhlig. Scalability of Microkernel-Based Systems[Dissertation]. Universitat Karlsruhe (TH), 2005.

[4] Alexander Warg, Software Structure and Portability of the Fiasco Microkernel [Dissertation]. Dresden: Technical University of Dresden, 2003

[5] Dietrich Claub. Implementation of the L4 Version x.2 ABI in the Fiasco Microkernel. Technical Report. Dresden: TU Dresden, 2004

[6] Intel Corp. Intel Architecture Software Developer's Manual, Volume 3: System Programming Guide, 2005.

[7] 王峥. 微机应用系统中的可靠性剖析[J]. 微电子学与计算机, 2006, 23(11): 74-77

[8] Andy Chou, Junfeng Yang, Benjamin Chelf, Seth Hallem, and Dawson Engler. An empirical study of operating systems errors. Proceedings of the eighteenth ACM symposium on Operating systems principles, Oct 2001.

[9] 赵立业，张 激，游 夏. 实时操作系统的性能分析和评估[J]. 计算机工程, 34(8): 283-285

[10] 陈幼雷，黄 强，沈昌祥. 操作系统可信增强框架研究与实现[J]. 计算机工程, 2007, 33(6): 12-14

[11] Härtig, Hermann; Hohmuth, Michael; Liedtke, Jochen; Schönberg, Sebastian (October 1997). "The performance of μ-kernel-based systems". Proceedings of the sixteenth ACM symposium on Operating systems principles: 66–77. ISBN 0-89791-916-5

[12] Jochen Liedtke. Lava Nucleus (LN) Reference Manual (486, Pentium, PPro) Version 2.2. 1998.

[13] L4 Document. Available from URL:http://os.inf.tu-dresden.de/L4

[14] Fiasco Document. Available from URL: http://os.inf.tu-dresden.de/fiasco/

[15] Operating Systems Research Group. L4Env: An Environment for L4 Applications. Technical report. Dresden: TU Dresden, 2003

[16] E.G.H. Schierboom. Verification of Fiasco's IPC implementation

[Dissertation]. Nijmegen: Radboud University, 2007.

[17] J.Liedtke. Improving IPC by kernel design. Proceedings of the 14th ACM Symposiumon Operating System Principles(SOSP), pages175–188, Asheville, NC, December 1993. 1

[18] Hendrik Tews. Case study in coalgebraic specification: Memory management in the Fiasco microkernel [Dissertation]. Dresden: TU Dresden, 2000

[19] Bernhard Kauer. L4.sec Implementation - Kernel Memory Management [Dissertation]. Dresden: TU Dresden, May 2005. 1

[20] T. Li, D. Baumberger, D. A. Koufaty, and S. Hahn. Efficient Operating System Scheduling for Performance-Asymmetric Multi-Core Architectures. In SC '07, 2007.

[21] Intel Core2 Memory Hierarchy. Intel Corp. http://www.cs.uiuc.edu/class/sp08/cs433/notes/Intel_Core2_Mem.ppt

[22] Daniel Potts, Simon Winwood, Gernot Heiser. Design and implementation of the L4 microkernel for Alpha multiprocessors. Technical Report UNSW-CSE-TR-0201, University of New South Wales, Australia, February 10, 2002.

致 谢

首先我要感谢导师周玲玲副教授。在两年多的研究生生活中，无论是在课程学习还是论文研究上，周老师都给与了无私的关心和指导，本文的选题、研究、撰写、修改到完成更离不开她的关心和帮助。在此，我对她在学习、生活中无微不至的关心与帮助表示衷心的感谢！

同时，我要感谢徐国治教授在我学习的三年中对我的学习和生活上的关心以及作为一名长者对后辈的人生道路的循循善诱和语重心长。在实验室的三年无疑将成为我人生当中的一段珍贵经历。

此外还要感谢应忍冬老师对本论文的选题和内容提供了许多宝贵意见，此论文涉及的项目也是在他的带领下进行的。

感谢 Intel 公司的 Susie Li 和包益平工程师在论文相关的项目过程中提供的帮助和指导，回答了我在项目中的疑惑。

另外还要感谢项目组张易知同学，在学习研究上的相互交流讨论使我获益匪浅，获得了提高。

感谢实验室所有的师兄师姐师弟师妹，大家一起度过的快乐而美好的时光是我一生都将永远珍藏的宝贵回忆！

最后，也对一直支持我的朋友和亲人表示诚挚的谢意。

攻读学位期间发表的学术论文

1. 张荫芾、应忍冬、周玲玲，支持多核处理器架构的微内核操作系统设计，计算机工程，2009 年第 24 期（已录用）
2. 张荫芾、徐国治、周玲玲，微内核操作系统在嵌入式平台上的应用，电子产品世界， 2009 年第 3 期（已录用）

上海交通大学学位论文答辩决议书

姓　　名	张荫芾	学号	1060349015	所在学科	电子工程系电路与系统专业
指导教师	周玲玲	答辩日期	2009.1.16	答辩地点	电信群楼 1-435
论文题目	基于多核处理器架构的嵌入式微内核操作系统的研究与设计				

投票表决结果：5 /5 /5 （同意票数/实到委员数/应到委员数） 答辩结论：□通过 □未通过

评语和决议：

多核处理器架构在嵌入式领域的普及已经成为一种趋势，微内核操作系统非常适应嵌入式设备的需求，但是微内核操作系统尚无对多核处理器架构的支持。论文设计了一个适用于多核嵌入式平台的微内核操作系统，在选题上具有理论意义和实际应用价值。

论文提出将分离模型的概念应用到系统设计中，在此基础上，论文对操作系统的内存管理、线程调度、线程间通信和服务应用程序等方面进行了设计和实现。在线程间通信方面，论文设计了一个邮箱系统作为多核间的 IPC 机制，并通过代理线程的方式实现透明地调用，具有一定的创新性。邮箱系统充分利用了多核处理器架构共享 L2 Cache 的特点，性能分析证明此设计能提高跨核 IPC 的速度，进而提高整个系统的性能。

论文结构合理、内容丰富、图表完整、表达清晰，实验数据真实完备，很好地证实了论文的观点，表明作者具有较扎实的专业基础知识和解决实际问题的能力，论文达到了工学硕士学位的要求，答辩过程中回答问题正确，经答辩委员会表决一致同意通过其工学硕士学位论文答辩，并建议校学位评定委员会授予其工学硕士学位。

2009 年 01 月 16 日

答辩委员会成员签名	职务	姓名	职称	单 位	签名
	主席	徐国治	教授	上海交通大学	
	委员	徐雄	副教授	上海交通大学	
	委员	刘佩林	教授	上海交通大学	
	委员	周玲玲	副教授	上海交通大学	
	委员	宫新保	副教授	上海交通大学	
	委员				
	委员				
	秘书	张玫	工程师	上海交通大学	