

注：教师姓名后留有一个空格，后面填写教师职称。下面加下划线。

阅后删除此文本框。

**本科毕业论文**

**基于x86架构的微内段基址交换算法的优化**

|  |  |  |
| --- | --- | --- |
| **学生姓名** | **：** | 郑嘉滨 |
| **学号** | **：** | 171543346 |
| **学院** | **：** | 互联网金融与信息工程 |
| **专业** | **：** | 计算机科学与技术 |
| **指导教师** | **：** | 邓伟林 **职称：** |
| **提交日期** | **：** | 年 月 日 |

**本科毕业论文（设计）诚信声明**

本人郑重声明：所呈交的本科毕业论文（设计），是本人在指导老师的指导下，独立进行研究工作所取得的成果，成果不存在知识产权争议，除文中已经注明引用的内容外，本论文不含任何其他个人或集体已经发表或撰写过的作品成果。对本文的研究作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。

 学生签名：

时间： 年 月 日

**关于论文（设计）使用授权的说明**

本人完全了解广东金融学院关于收集、保存、使用学位论文的规定，即：

1.按照学校要求提交学位论文的印刷本和电子版本；

2.学校有权保存学位论文的印刷本和电子版本，并提供目录检索与阅览服务，在校园网上提供服务；

3.学校可以采用影印、缩印、数字化或其它复制手段保存论文。

本人同意上述规定。

学生签名：

时间： 年 月 日

摘 要

操作系统的发展至今已有超过60年的历史，从最初的无结构操作系统到宏内核再到微内核架构，计算机的应用场景和功能不断变化，操作系统也顺应着变化。 就目前而言，主流的操作系统还都是宏内核架构，这种架构在灵活性和安全性上都存在一定的缺陷，所以近年来学界和业界都将关注点放在了微内核架构上。根据目前微内核的发展趋势来看，在未来微内核存在着取代宏内核占据主流市场的可能。

本论文的主要内容分为两部分，第一部分是基于32位保护模式实现一个简单的微内核；第二部分是利用这个内核来验证改进后的段基址交换算法。在微内核的结构方面，实现了Bootloader、进程调度模块、中断处理模块和进程间通信模块；在实现的平台方面，介绍了x86架构中的硬件机制和特性，包括寄存器的拓展、内存分段机制、CPU运行的三个模式等。在完成这些内容后，分析了物理页面交换算法和段基址交换算法的缺陷，并提出“磋商”型段基址交换算法来改进原来算法的缺陷，然后在本文所实现的微内核基础上，利用消息传递机制对三种进程间通信算法进行性能比较。最终得出结论，本论文所提出的“磋商”型段基址交换算法能够有效的减少微内核在某种通信场景下的通信次数。

**[关键词]**：微内核；进程间通信；段基址交换算法

Abstract

**[Key Words]:** Microkernel; Inter-process communication; Segment base exchange algorithm

目 录

[摘 要 I](#_Toc67970188)

[Abstract I](#_Toc67970189)

[目 录 III](#_Toc67970190)

[1 绪论 1](#_Toc67970191)

[1.1 研究背景 1](#_Toc67970192)

[1.2 国内外研究现状 2](#_Toc67970193)

[1.3主要研究工作 3](#_Toc67970194)

[1.4 章节安排 3](#_Toc67970195)

[2 相关技术基础 4](#_Toc67970196)

[2.1微内核架构 4](#_Toc67970197)

[2.1.1 微内核架构简介 4](#_Toc67970198)

[2.1.2 微内核相对于宏内核的优点 6](#_Toc67970199)

[2.2 x86架构 7](#_Toc67970200)

[2.2.1 Intel 8086 7](#_Toc67970201)

[2.2.2 Intel 80386 8](#_Toc67970202)

[2.3 x86-64架构 10](#_Toc67970203)

[2.4 NASM与GCC 10](#_Toc67970204)

[2.5 Bochs虚拟机 11](#_Toc67970205)

[3 x86架构中内核的总体设计和实现 12](#_Toc67970206)

[3.1系统的整体结构 12](#_Toc67970207)

[3.2 Bootloader 15](#_Toc67970208)

[3.3 进程调度模块 18](#_Toc67970209)

[3.4 中断处理模块 21](#_Toc67970210)

[3.5 IPC模块 22](#_Toc67970211)

[4微内核中进程间的通信模型 27](#_Toc67970212)

[4.1 进程间通信（Inter-process communication, IPC） 27](#_Toc67970213)

[4.2 共享存储器系统（Shared-Memory System） 28](#_Toc67970214)

[4.3 消息传递系统（Message Passing System） 28](#_Toc67970215)

[4.4 管道通信系统（Pipe Communication System） 29](#_Toc67970216)

[4.5 客户机-服务器系统（Client-Server System） 30](#_Toc67970217)

[4.6 物理页面交换算法与段基址交换算法 30](#_Toc67970218)

[5 “磋商”型SBEA（N-SBEA）与性能验证实验 32](#_Toc67970219)

[5.1 PPEA和SBEA存在的问题 32](#_Toc67970220)

[5.2 优化策略 33](#_Toc67970221)

[5.3 SBEA与N-SBEA的实现 33](#_Toc67970222)

[5.4 性能验证实验与结果分析 36](#_Toc67970223)

[6全文总结与展望 37](#_Toc67970224)

[6.1 全文总结 37](#_Toc67970225)

[6.2 存在的不足 38](#_Toc67970226)

[6.3 未来展望 38](#_Toc67970227)

[参考文献 39](#_Toc67970228)

[致 谢 40](#_Toc67970229)

[附录 内容名称 41](#_Toc67970230)

**基于x86架构的微内核操作系统的实现**

1 绪论

1.1 研究背景

操作系统（Operating system，OS）是一组主管并控制计算机操作，运用和运行硬件、软件资源和提供公共服务来组织用户交互的相互关联的系统软件程序，同时也是计算机系统的内核与基石。第一个操作系统（同时也是第一个批处理操作系统）是20世纪50年代中期由General Motors开发的，用在IBM 701上**[1]**。

在第一个简单批处理操作系统出现后，操作系统的发展经历了多道批处理系统、分时系统、实时系统、分布式操作系统等阶段，不同类型的操作系统有不同的应用场景。而在早期开发操作系统时，设计者通常只将关注点放在如何得到高效率上，对系统的设计缺乏首尾一致的思想**[2]**，导致随着系统的变大，整体呈现处混乱和安全性低的态势。不仅如此，随着代码的剧增，操作系统的可维护性大大降低，给维护人员增加负担。于是，操作系统结构化的思想应运而出。

操作系统的结构先后经历了无结构OS、模块化结构OS、分层式结构OS、微内核结构OS。现代的操作系统也可以大致分为宏内核OS和微内核OS。从当代主流操作系统的内核架构中，我们可以很清晰的看到，宏内核在当今的主流的操作系统架构中占据着绝对的优势。但是，如今操作系统的各种问题的出现，反过来暴露了宏内核的各种弊端**[3]**。宏内核一般包含了操作系统的大部分功能模块，代码高度集成，某个模块出现一个bug都有可能使整个系统崩溃。与之相对的，微内核架构的基本原则是机制跟策略相分离，把策略放置在操作系统较高层次，而机制置于微内核之中，也仅把系统最为核心、最为基础的模块放置于微内核中。同时，微内核操作系统给在外部空间的服务分配私有、独立的地址空间。这样，各服务间的隔离性得到保障，一个服务的崩溃不会对其他服务的正常工作产生影响。然而凡事都有两面性，微内核将部分服务放在用户空间必然导致进程之间的通信（Inter-process communication，IPC）频率的剧增，同时IPC带来的是频繁的上下文切换，即不断切换用户态和内核态。经过统计，执行微内核操作系统的绝大多数相关方法在用户态会产生一定的性能流失，约5%-10%左右**[4]**。但如今的计算机硬件性能与日剧增，考虑到微内核架构给整个系统带来的安全性，性能上的一点损失在一些条件下完全可以接受。

综上所述，相对于宏内核的高度集成和臃肿的设计架构，微内核优美的结构具有明显的优势，而这种优势随着当前嵌入式物联网技术的飞速发展愈发明显。微内核架构明显是未来的操作系统架构的趋势。

1.2 国内外研究现状

RC4000可能是第一个具备微内核架构思想的系统。这一系统可以用逐层添加操作系统的方法逐步扩充，以适应在进程调度与资源分配方面格式各样的不同需求。微内核术语早在1981年之前就出现了**[3]**。宏内核内高度集成的易变更的设备代码、协议、文件系统服务等使得宏内核随着规模的增大需要开发人员对这个内核代码进行复杂而又细致的管理，同时在某一项服务需要更升级更新时需要对整个内核的代码进行重新编译。为了克服这些问题，微内核思想由此产生。通过将一些需要频繁更新和修改的服务像其他用户服务一样，在用户空间实现这些服务，同时采用策略与机制分离的原则。机制放置于内核，策略放置于用户空间。

如今，经过数十年的发展，微内核已经发展到了第三代微内核。公认的第一代微内核操作系统是Mach，是由卡内基梅隆大学开发，为了用于操作系统研究，特别是分布式与并行运算上。但由于Mach初代内核提供了较多的服务，第一代Mach内核也被称为“胖微内核”。但Mach却有良好的跨平台性，Mach设计伊始将多处理器支持纳入考虑，是为千种处理器而设计的，移植到其他体系结构很容易。第二代微内核以L3和L4为代表，只提供最基本的操作系统服务并且注重性能优化。最显著的一点在于IPC中的优化，在同样的系统中Mach需要114毫秒来发送即便是最短的消息，而L3可以用少于10毫秒的时间来发送同样的消息**[5]**。L4内核是在L3的基础上发展起来的，对L3内核中具有的特性再次进行缩减， 就像L4/MIPS作者们所表述的：“一项特性当且仅当安全需要它在特权模式被实现时才应该在微内核里” **[6]**。由于它十分简单，L4很快被证明是具有高可移植性的。第三代微内核则主要重视系统的安全性问题等，其主要代表有：seL4、Fiasco、NOVA等**[3]**。

微内核不仅在学界取得了广泛的关注和影响，在业界也有逐渐颠覆宏内核的趋势。安卓(Android)操作系统是一个基于Linux内核与其他开源软件的开放源码的移动端操作系统。据操作系统行业数据分析显示：2019年手机端Android操作系统市场份额达68.63%。然而Google并没有止步于Android，在2016年之前Google就已经着手研发用于嵌入式系统的全新操作系统Google Fuchsia，这款OS不再是基于Linux这类宏内核，而是基于一个全新的名为Zircon的微内核；无独有偶，华为在2019年8月9日的开发者大会上，正式发布了鸿蒙操作系统（Harmony OS），一款全新面向全场景的分布式操作系统，而这款操作系统也是基于微内核架构的。

综上所述，由内核架构在学界和业界的发展历史可知，微内核架构的操作系统的已经是大势所趋，宏内核开始有被摒弃的迹象。尽管以当前市场上流行的操作系统来看，似乎印证了Linus Torvalds在那场和Andy Tanenbaum（MINIX之父）的口水仗所说的，微内核仅具有结构上的优美，而在性能上却完全不及宏内核。但日后，随着学界和业界对微内核的瓶颈问题——IPC的优化，同时凭借着微内核的良好的可移植性和可维护性，微内核还是非常有可能成为未来的操作系统的主流架构。

1.3主要研究工作

本论文的主要工作内容如下：

（1）通过查阅各种制作操作微内核的书籍，制作了一个基于x86架构（80386）的简易微内核，并实现了基本的bootloader、进程调度、IPC、内存管理、中断处理五大基本模块；

（2）对当前常用的IPC模型进行讨论，说明它们各自的优缺点；

（3）学习并研究了文章**[3]**的物理页面交换算法（Physical Pages Exchange Algorithm，PPEA）和段基址交换算法（Segment Base Exchange Algorithm，SBEA），并发现SBEA在某种场景下的缺陷，提出利用“磋商”的方式完善SBEA算法。

1.4 章节安排

本论文的主要工作分为两部分，第一部分是实现一个简易的基于x86架构的微内核，第二部分是对SBEA进行某些应用场景下的改进：

第一章为绪论，介绍了操作系统、宏/微内核架构、微内核的发展历史以及近年来国内外发展现状，同时概括了本论文的主要创新点；

第二章为相关技术基础，介绍了微内核架构及其特点、x86架构中的鼻祖8086和第一次引入32位保护模式的80386，此外对本次微内核的开发工具进行了概述；

第三章为微内核中进程间的通信模型，介绍了当前常用的IPC模型，包括共享存储器系统、消息传递系统、管道通信系统、客户机-服务器系统等；

第四章为x86架构中内核的总体实现，该章节先从整体上描述了系统所实现的功能，再对bootloader、进程调度、IPC、内存管理、中断处理五个模块进行了实现；

第五章为对SBEA算法的改进，开始先分析出SBEA算法在某些场景下存在的问题，接着再提出本论文对算法的改进策略；

第六章为性能验证试验与结果分析，利用本论文所实现的微内核对SBEA和改进后的SBEA进行极端条件下的性能试验，同时得出结果并分析；

第七章是结论，该章节对本论文进行总结，指出本论文工作中的不足之处并对未来进行了展望。

2 相关技术基础

2.1微内核架构

2.1.1 微内核架构简介

现代的操作系统的内核通常分为三种：微内核（Micro kernel）、宏内核（Monolithic kernel）和混合内核（Hybrid kernel）。20世纪五十年代中期到后期，操作系统的开发者很少考虑结构问题，也不具有构造大型操作系统的经验。在一些单体系结构的操作系统中，任何一个过程都可以调用任何其它过程。因此，在早期操作系统的规模不大时，这种结构的缺点如缺乏扩展性、可移植性、灵活性等问题没有明显的暴露出来。然而随着操作系统代码量的增加和规模的增大，这种缺乏结构的方法就无法支撑了。

结构化的操作系统设计方法随之出现，首先出现的就是分层的操作系统，如图2.1所示：



图2.1 分层的内核

但是分层的结构也存在一些问题。由于相邻层之间存在许多交互，且每一层处理相当多的功能，使得一层中的大的变化会对相邻层（上一层或下一层）中的代码产生巨大的影响。其结果是，在基本的操作系统上很难通过实现增加或减少一些功能实现一个专用的版本。另外，相邻层之间的高频交互使得安全性很难的到保证。

第一代微内核结构出现于20世纪80年代，是由Carnegie Mellon University的Avie Tevanian和Richard Rashid主导开发的Mach微内核。微内核的基本原理是，只有最基本的操作系统功能才能放在内核中。非基本的服务和应用程序在微内核之上构造，并在用户态下执行。尽管什么应该在微内核中、什么应该在微内核外，不同的设计有不同的分界线，但是共同的特点是许多传统上属于操作系统一部分的功能现在都是外部子系统，包括设备驱动程序、文件系统、虚存管理程序、窗口系统和安全服务。微内核结构采用水平分层的结构代替了传统的纵向分层结构，如图2-2所示：



图2.2微内核

微内核架构大多用在嵌入式操作系统中，包括：Mach，QNX，Sel4等标准微内核。其次之外，还有Windows NT，Mac OS X这类变种微内核。它们为了追求性能，将需要具备特权的服务组件放进核心空间，违反了微内核的基本设计原则，更接近宏内核的设计方式。这被称为混合内核，是介于微内核与宏内核之间的一种类型。

2.1.2 微内核相对于宏内核的优点

微内核设计的基本思想是简化内核功能，使在内核空间之外的用户空间尽可能多地实现系统服务，同时在系统服务之间加入安全保护措施。于此相对的，在宏内核架构中用户服务和内核服务在同一空间中实现，内核管理着进程调度，内存管理，文件管理和系统调用等模块的工作，由于用户服务在内核服务被实现在同一空间中，这样执行速度上要比微内核快。然而宏内核的劣势也显而易见，那就是内核中的某个服务崩溃，整个内核也崩溃了。另一点，想要在内核中添加新的功能就意味着内核的各个模块需要做相应的修改，因此其扩展性弱。

尽管微内核由于系统模块的分离导致性能上稍不如宏内核，但是微内核依然具有宏内核所不具备的巨大优势。微内核系统结构的核心优势有：（1）内核体积小，方便移植和进行维护；（2）进行拓展修改简易，这是因为微内核很多重要核心模块基本在核的外面完成；（3）系统使用人员很容易更新软件，是由于其仅是硬件的低级抽象，有很多其他模块的部分是到内核外面执行的，因此即使需要完成二次开发改造，也无需对内核再进行研发拓展；（4）可靠性比较高，这是因为每个核心模块都以离散部分模型存放在内核之外，当一个功能模块突然故障也不会干扰其他模块的正常运行。

但是，阻碍着微内核发展的恰巧也是微内核的优点所在之处。微内核架构本质上是一种客户-服务器模式（C/S Model），在微内核架构将众多系统服务放置到用户空间的同时，为系统服务间带来了地址空间边界，因此进程之间的通信（Inter-process communication，IPC）都需要经过内核再将消息发送给目标服务，这时就需要不断的切换用户态和内核态来实现系统。

总而言之，微内核架构所存在的一些问题可以归为以下几点：  
（1）进程或者服务间采用进程间通信的方式交换数据或者调用系统服务，而不是使用宏内核所使用的指针调用方式，这将造成额外的操作系统开销。  
（2）执行一些频繁使用的系统服务时，比如网络收发数据，造成频繁地上下文切换对操作系统来说也是一个负担。  
（3）由于系统服务高度模块化，系统服务之间存在大量的内存复制。  
（4）对互相之间存在的复杂调用关系的系统服务，难以合理地设计通信接口。  
（5）系统服务与内核在地址空间上分离，造成代码局部性差，降低了cache命中率。

2.2 x86架构

2.2.1 Intel 8086

8086是Intel公司第一款16位处理器，诞生于1978年，是整个Intel 32位架构处理器(IA-32)的开山鼻祖。8086体系架构由Stephen P. Morse设计，并在最后定案时得到Bruce Ravenel（8087体系结构设计者）的帮助。逻辑设计者是以Jim McKevitt与John Bayliss为首的硬件开发工程师团队。项目经理是William Pohlman。迄今为止8086的指令集仍然是PC机与服务器的基本指令集。

8086包含8个16位通用寄存器和4个16位段寄存器，通用寄存器分别是：AX、BX、CX、DX、SI、DI、BP、SP，而AX、BX、CX和DX可以分别拆成两个8位AH和AL、BH和BL、CH和CL、DH和DL使用，8086的内部结构如图2.3所示：

图2.3 8086CPU内部结构

8086的内存寻址采用的是分段模型。由于它的地址总线是20位，最多可以寻址1M大小的内存，但是段寄存器只有16位，所以在进行寻址时要将段寄存器向左移4位，再加上16位的段内偏移，最终构成段地址：偏移量的组合。由于段寄存器是16位的，在段不重叠的情况下，最多可以将1MB的内存分成65536个段，段地址分别是0000H、0001H、0002H … ,一直到FFFFH。同样在不允许段之间重叠的情况下，每个段的最大长度是64KB，因为偏移地址是16位，从0000H到FFFFH。在这种情况下，1MB的内存最多能划分成16个段，每段长64KB，段地址分别是0000H、0001H、2000H、3000H、…、一直到F000H。

2.2.2 Intel 80386

Intel 80386 发布于1985年，是以1978年8086处理器为基础发展起来的。Intel 80386是第一款具有32位的保护模式的CPU，其将PC由16位的时代带入到了32位的时代。

80386具有三种工作模式：实模式（Real mode）、保护模式（Protected mode）、虚拟86模式（V86）**[12]**。其中的保护模式是本设计内核工作的主要模式。所谓的保护模式是一种与80286系列和之后的x86兼容的CPU的运行模式。保护模式有大量的新特性，如存储器保护、标签页系统以及硬件支持的虚拟内存，能够增强多任务处理和系统稳定度，现今大部分的x86操作系统都在保护模式下运行，包括Linux、FreeBSD、以及微软Windows 2.0和之后的版本。

在80386中，它对8086的一部分寄存器进行了拓展，由16位寄存器拓展成为32位寄存器同时增加了两个段寄存器GS和FS。它将AX、BX、CX、DX、SI、DI、BP、SP分别拓展成32位的EAX、EBX、ECX、EDX、ESI、EDI、EBP、ESP。段寄存器位数保持不变，当其中保存的不再是段机制，而是保存与全局描述符表（Global descriptor table，GDT）或者局部描述符表（Local descriptor table，LDT）相关的一个索引及属性，被称为段选择子（Selector）。

此外，每个段寄存器还包括一个64位不可见的部分，称为描述符高速缓冲器。Intel 80386的内部结构如图2.4所示：



图2.4 80386CPU内部结构

80386的地址线有32根，而在段描述符中段基址也是32位，所以理论上来说80386对内存的访问不再需要分段，但是为了向后兼容8086等x86架构的处理器，80386处理器还是需要以分段的方式访问内存。但是它也提供了一种变通的方式，即把整个4G内存看成一个段，段基址是00000H。在这种情况下可视为不分段，即平坦模型（Flat Mode）。

80386，以及后续的32位处理器都兼容实模式，可以运行实模式下的8086程序。在计算机一开始加电的时候，CPU的状态还是处于实模式下，此时它相当于一个非常快速的8086 处理器。在经过一系列的设置后，CPU才从实模式跳入到保护模式。

2.3 x86-64架构

x86-64（又称x64）最初由AMD在1999年设计，这个基于x86的64位架构向后兼容与16位及32位的x86架构，现在Intel称这个架构为Intel 64。Intel 64 与IA-64并不是同一个架构，Intel 64所用的x86-64是IA-32指令集的延伸，而IA-64则是另一款独立的架构，没有任何IA-32的影子**[14]**。x86-64理论上来说可以访问16EB的虚拟地址空间，但是目前的实现机制并不允许整个16EB的虚拟地址空间都被使用，理由是在可预见的未来，大多数操作系统不会用到如此巨大的地址空间，因此，x86-64架构只允许48位的地址宽度，即只能访问最大为256TB的虚拟地址空间，高16将作为符号拓展（将第47位数值扩展至第63位，即全0或全1），此种格式的地址被称为Canonical地址**[15]**。x86-64相比于IA-32增加很多特性，例如分页机制支持4KB、2MB和1GB的物理页容量，并支持多层级页表来结构化线性地址空间，cr3不再是页目录表的基址而是顶层页表的物理基地址，x86-64架构的页表地址映射过程如图2.5所示。

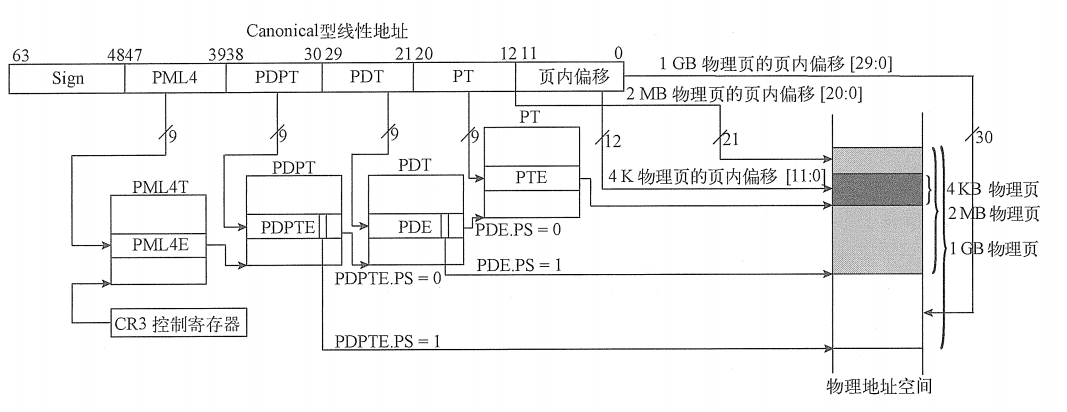


图2.5 x86-64架构页表地址映射过程

2.4 NASM与GCC

Netwide Assembler 是一款基于Intel x86架构的汇编与反汇编工具。它可以用来编写16位、32位（IA-32）和64位（x86-64）的程序，其所采用的是x86汇编语法的变体而不是AT&A语法。NASM也被用来编译本毕业设计的内核汇编代码，所采用的的版本是 2.10.07。

GNU编译器套装（GNU compiler collection，GCC）是一套编程语言编译器，以及GPL及LGPL许可证所发行的自由软件，也是GNU计划的关键部分，也是GNU工具链的主要组成部分之一。GCC原名为GNU C语言编译器（GNU C compiler），因为它原本只能处理C语言，后经过拓展，变得可以处理C++、Fortran、Pascal、Objective-C、Java等一系列主流的编程语言。有别于局限于特定系统与运行环境的编译器，GCC在所有平台上都使用同一个前端处理程序，产生一样的中介嘛，因此该中介码在各个其他平台上使用GCC编译，有很大机会可得到正确无误的输出程序。

GCC被用于编译本毕业设计的内核的C语言代码，所采用的版本是4.8.5。

2.5 Bochs虚拟机

Bochs 是一款开源的，能够用来模拟x86硬件平台的虚拟机。Bochs的典型应用是提供x86 PC的完整仿真，包括x86处理器（包括保护模式）、硬件设备、和存储器。这种特点使得能够在主机上模拟运行和调试操作系统的内核。

Bochs用于模拟运行和调试本毕业设计的内核，所采用的的版本是2.6.8。Bochs的配置文件bochsrc如下所示：

msg: 32

romimage: file-../bios/BIOS-bochs-latest

vgaromimage: file=../bios/VGABIOS-lgpl-latest

boot: floopy

log: bochsout.txt

mouse: enabled=0

cpu: ips=15000000

clock: sync=both

keyboard: keymap=../keymaps/x11-pc-us.map

3 x86架构中内核的总体设计和实现

3.1系统的整体结构

本论文旨在实现一个简单的微内核来验证改良后的段基址交换算法（Segment Base address Exchange Algorithm，SBEA）算法性能的优越性。整个微内核大致可分为五个部分，引导加载程序（bootloader）、进程调度模块、IPC模块、中断处理模块。其中引导加载程序没有遵循Multiboot规范，仅仅是根据特定的平台（80386）来编写的一个bootloader程序；进程调度模块并未采用复杂的进程调度算法，只是利用时钟中断进行切换进程，故本质上是一个时间片轮转调度算法，但是由于在设计时将调度算法独立写成pcbSchedule()函数，故具有非常好的扩展性；中断处理模块负责在CPU进入保护模式BIOS中断无效后，完成建立中断描述表（Interrupt Descriptor Table，IDT）、保护程序断点和程序现场以及转入中断服务程序，其中中断向量号为0x90被用于系统调用的中断向量号；IPC模块首先基于消息传递机制建立复制型的IPC，即通过系统调用sendrec(int function, int src\_dest, MESSAGE\* pMessage)指定sender或者receiver和所要接收或传送的数据pMessage。当receiver得知sender已经处于等待传输状态时（由PCB中的pFlags成员表示，宏SENDING表示等待传输状态，宏RECEVING表示等待接收数据状态，处于这两种状态的进程都将被阻塞）receiver将pMessage所指向的数据复制到自己的进程空间中，实现数据的传输。之后，IPC模块将会建立SBEA以及“磋商”型SBEA（N-SBEA）的IPC机制，进而验证N-SBEA的性能优越性。本论文实现的微内核的模块组成如图3.1所示：

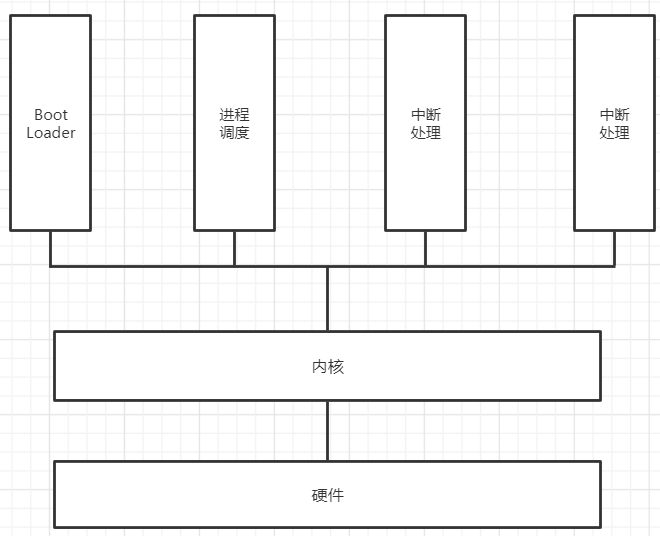


图3.1

为了使每次将Loader和内核写入软盘时不需要考虑写入位置和文件大小，将1.44MB的软盘制成FAT12的格式，使得Linux可以识别并挂载。FAT12的整体结构如图3.2所示：



图3.2 FAT12结构

创建一个FAT12header.inc文件来包含FAT12所需要的头部信息如下所示：

BS\_OEMName: db 'Virux ' ;manufacture name, must be 8 bytes

BPB\_BytsPerSec: dw 0x200 ;how many bytes in each section

BPB\_SecPerClus: db 0x01 ;how many section in each cluster

BPB\_RsvdSecCnt: dw 0x01 ;how many section boot section occupies

BPB\_NumFATs: db 0x02 ;how many FAT table totally

BPB\_RootEntCnt: dw 0xe0 ;how many files in root directory

BPB\_TotSec16: dw 0xb40 ;total section

BPB\_Media: db 0xf0

BPB\_FATSz16: dw 0x09 ;how many section in each FAT

BPB\_SecPerTrk: dw 0x12 ;how many section in each track

BPB\_NumHeads: dw 0x02 ;header number

BPB\_HiddSec: dd 0x0

BPB\_TotSec32: dd 0x0

BS\_DrvNum: db 0x0

BS\_Reserved1: db 0x0

BS\_BootSig: db 0x0

BS\_VolID: dd 0x0

BS\_VolLab: db 'Virux01 '

BS\_FileSysType: db 'FAT12 '

3.2 Bootloader

在CPU完成上电自检（Power On Self Test，POST）后就对第0磁头第0柱面第1扇区的最后两个字节进行检查，检查是否为0x55和0xaa。如果是这两个字节，CPU则确定这个扇区的内容是主引导记录（Master Boot Record，MBR），并将扇区的内容载入内存0x7c00处，再把CPU的控制权交给MBR。

由于引导扇区的大小限制在512B，但在把控制权交给内核前还需要完成诸如在根目录区寻找内核位置、载入内核、将内核调整到适当的位置、软驱复位、开启分页机制、将CPU跳入保护模式等一些工作，显然区区512B的扇区是不够的。所以引导扇区所完成的工作仅仅是在软盘的根目录区中寻找loader，并把loader载入内存中的0x9000:0x100处，再将控制权交给loader，由loader负责设置硬件环境和参数、寻找内核并载入内核。引导扇区完成的工作如图3.3所示：



图3.3 引导扇区工作流程图

进入loader后第一件事就是设置段寄存器和SP栈指针。SS段寄存器设置为0x9000，SP设置为0x100，这样loader的栈就是靠着loader起始位置向下扩展。由于在loader中要跳入保护模式，故需要建立全局段描述表（Global Descriptor Table, GDT），GDT利用自定义的宏Descriptor定义段描述符在/SourceCode/booter/Loader.asm的开头处。Loader所需要做的第二件事是在软盘的根目录检查是否有名为KERNEL.BIN的文件，若有则调用@ReadSection函数将内核载入到0x8000:0x00。@ReadSection定义在/SourceCode/booter/include/RealModeLib.inc中，其关键是利用入口参数调用0x13号中断读软盘。第三，完成载入内核后，loader调用0x15号中断检查内存的分布，在Bochs虚拟机中模拟出来的硬件环境的内存分布如图3.4所示：

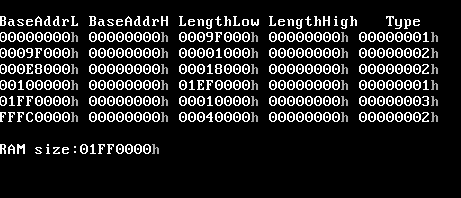


图3.4内存分布

最后一个Type字段具有如下含义：Type=1是AddressRangeMemory 表示这个内存段可以被OS使用的RAM；Type=2是AddressRangeReserved表示这个地址段正在被使用，或者被系统保留所以一定不要被OS使用；Type=其他则未定义**[11]**。得到内存分布情况是为了后面开启分页机制做准备，因为在32位环境下，地址线有32条可以表示4GB大小的内存，如果进行对等映射则需要4GB/4KB=1M个页表项，每个页表项大小为4B，所以单单是页表项就需要占据4MB大小的内存。然而我们实际可用的内存根据图3.4只有大约512MB，故只需要512KB的页表空间，每一张页表最大为4KB，所以只需要128张页表。页目录表同页表类似，最大为4KB，每一个条目4B，所以每一个页目录表最多可包含1K个页目录条目（Page-Directory Entry，PDE）。综上，分页机制只需要额外花费大约512KB+4KB=516KB的内存空间即可完成，远远小于原来的4MB。

Loader在实模式下所做的第四件事就是进入保护模式，但在此之前需要完成加载全局描述符表寄存器、关闭中断、打开第20条地址线、设置控制寄存器cr0以及利用一个特殊的跳转jmp dword 跳入保护模式。

进入保护模式后，按照惯例，首先要做的就是设置保护模式下的各个段寄存器。此时，段寄存器保存的不再是段基地址，而是段选择子，一个选择子指向段描述符的一个表项。此后，便是利用函数@SetupPaging开启分页机制，这个函数的关键设置cr3控制寄存器，该32位寄存器保留页目录表的基址。在此，将页目录表放置在0x10000:0x00处，并将所有页表连续存放在以0x10100:0x00开头的内存地址处，同时映射方式是对等映射。

最后，调用@InitKernel函数将内核从0x8000:0x00放到0x3040:0x00处。这里涉及到内核的文件格式问题。首先将所有内核文件用gcc或者nasm单独编译成.o文件，之后再利用链接器ld将所有的.o文件进行链接，链接形成一个可执行文件，这个文件的格式是ELF（Executable and Linkable Format）。由于内核这个可执行程序对所有的.o文件都保留有一个副本，显然，这个链接过程是静态链接。由于内核最终形成的是一个ELF格式的可执行文件，函数@InitKernel实际上就是根据ELF文件格式对内核进行初始化。ELF文件结构如图3.5所示：



图 3.5 ELF文件结构

其中Program header的内容用于描述段中内容如何放置内存中，包括段在内存中的基址、段的长度、段的在内存中的对齐方式等。program header的所有字段如表4.1所示：

表3.1 program header 内容

|  |  |
| --- | --- |
| field name | meaning |
| p\_type | type of segment described in current program header |
| p\_offset | the offset of the first byte of segment in |
| p\_vaddr | virtual address of the first byte in memory |
| p\_paddr | physical address |
| p\_filesz | size of segment in file |
| p\_memsz | size of segment in memory |
| p\_flags | flag about segment |
| p\_align | the way how the segment aligns in memory |

Loader便是根据program header字段的内容将内核文件移动到目的地址（0x3040:0x00）处，并最终利用一个远跳转跳入内核，把CPU的控制权交给内核。至此，loader的任务完成，后续的任务将交给kernel完成。

3.3 进程调度模块

本论文所写的内核是一个多进程系统，能够通过进程调度模块以一定的策略在不同进程之间进行切换。处于简单考虑，本内核所采用的的抢占式（Preemptive Mode）的基于时间片的轮转（Round Robin，RR）调度算法。所谓的“抢占式”是指允许调度程序根据某种原则，去暂停某个正在执行的进程，将已分配给该进程的处理机重新分配给另一进程**[2]**。

为实现进程的管理，在/SourceCode/kernel/include/process.h中定义出进程的结构体PCB，进程结构体如下所示：

typedef struct processControlBlock {

stackFrame stackframe;

u16 ldtSelector;

Descriptor ldt[LDTSIZE];

u32 processID;

int pFlags;

MESSAGE\* pMessage;

pMessage=0

u32 pReceivefrom;

u32 pSendto;

u32 hasIntMeg;

struct processControlBlock\* sendingQueue;

struct processControlBlock\* nextSending;

int ttyIndex;

char processName[32];

}PCB;

成员变量stackframe用于保存程序断点和CPU现场，主要的内容是各个段寄存器和通用寄存器。ldtSelector用与指向GDT中的一个表项，每一个进程都在GDT中都有一个LDT的全局描述符。数组ldt是局部段描述符表（Local Descriptor Table， LDT），它的功能与GDT类似，但不同在于它体现出了32位保护模式的思想，即它隔绝了各个进程的可访问空间。不同进程具有不同的LDT，在各个LDT包含的段不重叠的情况下，每一个进程都只能访问访问自己的LDT的段所描述的内存空间，无法访问其他进程的LDT段描述符所代表的内存空间，保证了某个进程不会被其他恶意的进程破坏。processID是系统全局唯一的进程标识，在内存加载进内存时会被分配唯一的标识。其他成员变量将在后续模块中详细说明。

为了对时间片的大小进行控制，需要对时钟中断发生的间隔进行控制。时钟中断产生的机构通常称为PIT（Programmable Interval Timer），一个典型的芯片就是Intel 8253。8253具有三个计数器，其中计数器Counter0连接在可编程的中断控制器（Programmable Interrupt Contorller，PIC）的IRQ0引脚上，以固定的频率发出时钟中断，8253的各个计数器的功能如表3.2所示：

表3.2 8253芯片计数器

|  |  |
| --- | --- |
| 计数器 | 作用 |
| Counter0 | 输出到IRQ0，以便每隔一段时间让系统产生一次时钟中断 |
| Counter1 | 通常被设为18，以便大约每15μs做一次RAM中断 |
| Counter2 | 连接PC喇叭 |

计数器的工作原理是：它具有一个输入频率，在PC上通常是1193180Hz。在每一个时钟周期（CLK cycle），计数器值会减1，当减到0时，就会触发一个输出。由于计数器是16位的，所以最大值是65535，因此默认的时钟中断的发生频率就是1193180/65536≈18.2Hz，也即大约54.9ms发生一次时钟中断。假设要将时钟中断的间隔时间设置为10ms，则要写入Counter0的数值是1193180/100≈11931。写8253计数器的端口如表3.3所示：

表3.3 8253的操作端口

|  |  |
| --- | --- |
| 端口 | 描述 |
| 0x40 | 8253 Counter0 |
| 0x41 | 8253 Counter1 |
| 0x42 | 8253 Counter2 |
| 0x43 | 8253模式控制寄存器（Mode Control Register） |

进程调度模块的核心是写在/SourceCode/kernel/process.c中的函数pcbSchedule()。在时钟中断发生时，首先进行中断处理，之后转入中断服务程序clockService()，再在clockService()中调用进程调度算法pcbSchedule()，对全局变量PCBready进行修改。决定运行那一个进程之后，就根据PCB结构体的成员变量stackframe对进程的断点和现场进行恢复。进程调度模块的大致流程图如图3.6所示：



图3.6 进程调度模块流程图

3.4 中断处理模块

当CPU处于实模式，中断实际上由BIOS进行处理。BIOS将256个中断服务程序的入口地址放在最低1KB处形成中断向量表（Interrupt Vector Table，IVT），也即0x00000~0x03FFF处。当发生硬件中断或者执行中断调用指令int时，就会根据中断号和入口参数调用BIOS中断。CPU跳入保护模式后，IVT失效，需要自行建立中断描述表（Interrupt Descriptor Table，IDT）和编写中断服务程序。在IDT中不仅保存着描述中断门的中断门描述符，而且还可以有任务门描述符和陷阱门描述符，利用中断门描述符可以实现系统调用，通过系统调用使得在低特权级（如Ring3）的用户进程可以获得内核的服务。在本系统中，系统调用的中断向量号是0x90.

中断处理模块包含两大部分，一部分是对中断的处理；另一部分是对异常的处理。中断分为外部中断和软中断，通常用来处理处理器外部的随机触发的事件，如键盘的输入、屏幕的输出等；异常包括Fault、Trap和Abort三种类型，通常是由于指令执行时发生错误而发生，例如特权级为3的段访问特权级为0的段，这将会触发#GP的Fault型异常。

外部中断（也叫硬件中断）主要由8259A可编程的中断控制器（Programmable Interrupt Controller，PIC）（如图3.7所示）触发，分为可屏蔽中断和不可屏蔽中断，分别由8259A的INTR引脚和NMI引脚进行接收。当CPU中的eflags寄存器的第9位IF=1时，CPU响应可屏蔽中断，反之则不响应。

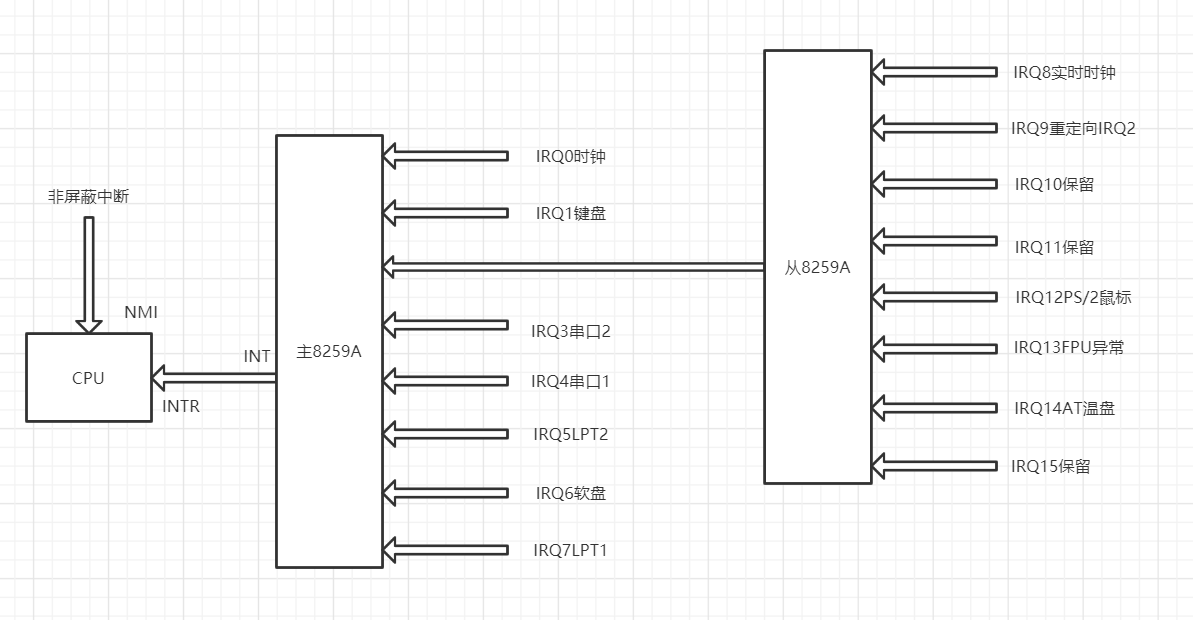


图3.7 8259A结构

本系统对中断处理模块的设计如下：对所有异常进行集中的统一处理，即将发生异常的指令处的CS、EIP和eflags的值打印在屏幕上，再调用hlt停机指令；而对于8259A中的15个硬件中断，只编写时钟中断和键盘的中断服务程序，其他中断也进行统一的处理。所有已处理的异常和中断的中断号和含义可在附录中的表（1）查看。

3.5 IPC模块

由于本论文实现的内核属于微内核架构，因此两个进程间的通信不能通过系统调用直接将数据由一个进程空间复制到另一个进程的空间，需要建立一套消息传递机制，使得用户进程（Ring 3）可以进入内核态（Ring 0）中，并由内核进行数据传输。微内核中的进程的通信可以被分为三个阶段：进入内核空间的模式转换、消息传递过程以及返回用户空间时的模式转换**[13]**。消息传递机制是微内核的区别与宏内核的关键特性。而且，由于微内核将许多任务分离成处于用户空间的用户进程，基于消息机制的IPC的发生频率极高，因此微内核中的IPC机制是通常是限制微内核性能的瓶颈。对IPC算法的研究和优化具有重要的意义。

本系统建立系统调用sendrec(int function, int src\_dest, MESSAGE\* pMessage)用于进行消息的传送，其功能号是3。该系统调用的核心函数时位于SourceCode/kernel/syscall/sendrecService.c的sendrecService(int function, int src\_dest, MESSAGE\* pMessage, PCB\* pPCB)函数，它通过对参数function进行甄别，如果function=SEND，则调用函数sendMessage（PCB\* pPCB, int src\_dest, MESSAGE\* pMessage）；如果function=RECEIVE，则调用receiveMessage（PCB\* pPCB, int src\_dest, MESSAGE\* pMessage），SEND和RECEIVE是定义在SourceCode/kernel/include/const.h中的两个宏。结构体MESSAGE如下，其中的有效数据部分是联合体u，大小为108B。

typedef struct {

int source;

int dest;

int type;

int reply;

PACKAGE\* pPackage;

union {

struct mess1 m1;

struct mess2 m2;

struct mess3 m3;

}u;

}MESSAGE;

又因为这是一个同步的消息传递机制，若sender调用sendMessage后，其目的接收方receiver必须也调用receiveMessage来接受message，否则，sendre将由于数据无法传输到目的进程导致sender由运行状态或者就绪状态转变为阻塞状态，无法被PCB调度模块所调度，倘若如此，PCB的成员pSendto将会被修改为目的进程的序号；同理，如果receiver调用了receiveMessage（）而没有sender传输数据给它，它也要由运行或就绪态变成阻塞态，并且PCB中的pReceivefrom成员会被修改为所要的sender的序号。进程的状态由PCB中的pFlags变量所表征，当pFlags=0时，表示处于运行或就绪态；当pFlags=SENDING或者RECEVING时，它则处于阻塞态。有一种情况需特殊处理，如果receiver等待多个sender传输数据给它，内核将会将这多个sender的PCB指针以无头结点链表的方式建立在receiver中的sendingQueue中，receiver的成员变量pReceivefrom此时可以修改为两种值，一种是特定的sender的序号，表示必须先接收这个sender的数据；另一种则是ANY（定义在头文件的宏），表示可以接收任意一个进程的数据。

通常情况下，一个完整的系统调用需要一个来回，用户进程向内核请求服务，然后内核需要返回给它。消息机制的本质是一个系统调用，因此一般来说，用户进程发送一个消息之后需要马上等待接收一个消息，以便收到内核（其实是某个系统任务）给它的返回值。这种发送-接收的行为被封装在函数send\_recv()（SourceCode/kernel/process.c）中，并提供一个BOTH的消息类型，当sender发送BOTH消息类型时，将会立刻等待一个返回的消息。

在性能方面，这个基于复制型的消息通信机制的主要耗时在于sendMessage() 和 receiveMessage()中的memcpy()函数，这个函数被利用汇编实现在SourceCode/kernel/lib/string.asm中，将传入的两个指针所指向的地址进行以字节为单位的直接内存写入。若A进程需要传送N B的数据给B进程，那么其时间复杂度是O(N)。

为了对基于消息传递机制的复制型IPC进行验证，利用两个进程（TestA和TestB）进行单向的数据传输，进程TestA的代码如下所示：

PUBLIC void TestA() {

int ticks = 0;

int number = 1000;

printf("In TestA now\n");

printf("Number=%d\n", number);

MESSAGE msg;

ticks = getTicks();

printf("tick1=%d\n", ticks);

for (int index; index < number; index++) {

resetMessage(&msg);

msg.u.m1.m1i1 = 100;

msg.source = 2;

msg.dest = 3;

msg.type = COPY;

send\_recv(SEND, 1, &msg);

do {

send\_recv(RECEIVE, 1, &msg);

} while (!msg.reply);

}

ticks = getTicks();

printf("tick2=%d\n", ticks);

while (1) {

delayInMilli(1);

}

}

因为本论文的着重点在于IPC机制的通信效率，因此进程的本身的构造设计的比较简单，不存在I/O操作，代码仅仅是利用系统调用与另一个进程进行通信。在利用send\_recv()向系统任务进程sys\_task发出通信请求后，TestA将会等待sys\_task发回一个message，并查看其中的成员变量reply，如果reply等于就开始准备下一次IPC。TestB的代码如下所示:

PUBLIC void TestB() {

printf("In TestB now\n");

MESSAGE msg;

MESSAGE response;

while (1) {

resetMessage(&msg);

resetMessage(&response);

do {

send\_recv(RECEIVE, 1, &msg);

} while (!(msg.type==COPY));

response.source = 3;

response.dest = 1;

response.type = ECHO;

send\_recv(SEND, 1, &response);

dataProcess(&msg);

}

while (1) {

delayInMilli(1);

}

}

它利用一个while循环不断接受message，并判断其中的type成员是否为COPY。如果是，一个ECHO类型的message将会被返送给sys\_task，sys\_task接收到这个ECHO报文后，最后发送一条回复message（reply成员变量被置为1）给TestA表示此次通信完成，可以开始下一次IPC。sys\_task代码如下：

case COPY: {

msg.source = 1;

msg.dest = dest;

msg.type = COPY;

send\_recv(SEND, dest, &msg);

do {

send\_recv(RECEIVE, dest, &msg);

} while (!(msg.type==ECHO));

msg.reply = 1;

send\_recv(SEND, src, &msg);

break

}

通信的全过程如图3.8所示：

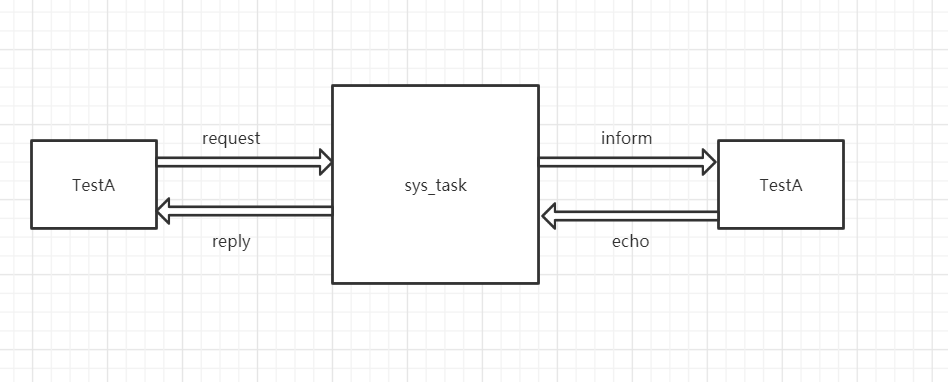


图3.8 复制型IPC全过程

对于SBEA以及N-SBEA算法的具体实现以及三个IPC通信机制的量化实验，将会在下个章节中进行详细的描述。

4微内核中进程间的通信模型

4.1 进程间通信（Inter-process communication, IPC）

IPC是指进程之间信息的交换**[2]**。微内核引入了地址空间的概念使得各个线程运行在各自独立的地址空间中。但不同进程必不可免地要进行数据交换、传递参数、发送消息或相互间执行同步操作，由于地址空间分离的原因，进程不能直接访问其他进程的地址空间的数据。这样,微内核必须提供相应的通信机制。

随着微内核的发展，进程通信机制已由早期的低级进程通信机制发展为能传送大量数据的高级通信工具机制。高级通信机制通常分为四大类：共享存储器系统、消息传递系统、管道通信系统、客户机-服务器系统。下面分别介绍这四种通信模型。

4.2 共享存储器系统（Shared-Memory System）

共享内存是需要通信双方在系统中开辟一块能够共同访问处于同种逻辑内存区域的一种通信媒介**[7]**。进程双方在利用共享内存通信时，同时需要采用信号量机制进行同步操作。微内核采用共享内存方式作为IPC机制需要经过4个阶段分别是创建、连接、访问以及释放。系统还需要为进程提供系统调用以便进程能够向内核传入相关的参数，比如申请共享内存的大小、系统对内存进行访问时需要执行信号量的锁定操作。互斥约束不属于共享内存机制的一部分，但必须由使用共享内存的进程提供。

共享存储器系统又根据所共享区域的类型分为基于共享数据结构的通信方式和基于共享存储区的通信方式**[2]**。第一种要求进程公用某些数据结构来实现通信，如在生产者-消费者模型中的有界缓冲区，这在具体的C语言编程时体现为全局变量。然而一个数据结构的大小有限，故每次不可能传送大量的数据，基于共享数据结构的通信方式实际上是一种低级通信。第二种基于共享内存区的通信方式是一种高级通信。内核为进程双方在内存中划分出一块共享存储区，进程通过对该内存区域进行读写来交换信息实现通信。同步控制当然也需要用到信号量机制。在进程完成通信后，需要将其归还给共享存储区。

4.3 消息传递系统（Message Passing System）

在这种模型中，进程不需要控制任何共享存储区和数据结构，只需要以格式化的消息（message）为单位，将数据封装进消息中，并利用内核提供的通信命令（或通信原语），完成进程间的的数据交换。这种方式隐藏了通信实现细节，使通信过程对用户透明化，降低了通信程序设计的复杂性和错误率，成为当前最广泛使用的一类IPC的机制**[2]**。这种消息封装的思想类似于计算机网络中，上一层协议将报文传给下一相邻层后，下一相邻层对上一层隐藏具体的传输细节，对于上一层而言，可以看做是点对点（Point to Point）传输。

内核一种常见的消息传递系统实现方案是实现一个消息队列。以两进程通信为例，队列两头连接着进程双方。当Receiver从没有消息的空队列中获得消息时，它会处于阻塞状态等待Sender将消息放入消息队列中；当Sender向已满的队列中放入消息时，它也会进入阻塞状态等待Receiver将消息从消息队列中取走。消息队列通信模型如图4.1所示：



图4.1 消息队列通信模型

4.4 管道通信系统（Pipe Communication System）

管道是指用于连接一个读进程和一个写进程以实现它们之间通信的一个共享文件，又名pipe文件**[2]**。管道通信机制始创于UNIX，Linux则沿用了UNIX的这个IPC机制。管道在创建时获得一个固定大小的字节数。当一个进程试图往管道中写时，如果有足够的空间，则写请求被立即执行；否则该进程被阻塞。类似地，如果一个读进程试图读取多于当前管道中的字节数时，它也被阻塞；否则读请求被立即执行。内核对管道强制实施互斥，即一次只能有一个进程可以访问管道。UNIX系统中的管道分为两种，匿名管道（PIPE）和命名管道（FIFO），分别工作在不同领域以完成进程的信息交换及共享**[8]**。

匿名管道也称为无名管道，它的存在是临时性的，该类管道在实际应用中经常使用到。在Linux系统中使用pipe系统调用创建，它可被具有血缘关系的进程共享存取，当所有的进程都结束使用某个无名管道时，内核便回收它。命名管道在Linux中则使用mkfifo或mknod创建，类似于一般的普通文件，它和匿名管道的数据结构及操作非常类似，区别在于FIFO提供了一个路径名，这样即使不存在血缘关系的进程只要能够访问该路径，就能够通过FIFO通信**[9]**。

4.5 客户机-服务器系统（Client-Server System）

前面所述的共享内存机制、消息传递机制、管道通信机制通常都是用于同一台计算机的IPC，而对于不同计算机间的进程通信通常采用的是客户机-服务器系统模型。该模型广泛应用于基于网络环境下的计算机间不同进程的通信。  
其主要的实现方式有三种：套接字（Socket）、远程过程调用和远程方法调用（Remote Procedure Call，RPC）:

1)套接字起源于20世纪70年代的BSD UNIX，是UNIX操作系统下的网络接口。一个socket就是标识出通信类型的数据结构，它包含了通信目的地址、通信使用的端口号、通信网络的传输层协议、进程所在的网络地址、以及针对客户或服务器提供的不同系统调用等，是进程通信和网络通信的基本构建**[2]**。套接字一般有两种类型：基于文件型和基于网络型，前者可以实现同一台机器下的IPC，后者则是实现异构网络环境下的IPC。

2）远程过程调用和远程方法调用主张通过屏蔽网络和设备差异，应用软件可以便捷的调用网络中另一应用所提供的对外接口**[10]**。RPC本质上是一种高度封装的网络层之上的应用协议栈，结合标准化IDL（Interface Description Language）接口，约束通信双方的请求和响应行为，提供网络透明化的虚拟本地调用方式**[11]**。

4.6 物理页面交换算法与段基址交换算法

物理页面算法（Physical Page Exchange Algorithm，PPEA）的本质就是进程请求内核将sender的IPC缓冲页面的线性地址与receiver的IPC缓冲页面的线性地址进行交换，由此，receiver即可不用通过复制的方式就能够访问到sender所要传送的数据。因为在通信开始之前，内核为sender和receiver都各分配一个IPC缓冲页面，在通信前后双方都无法读取对方的地址空间中的缓冲页面内容，保证了sender和receiver的独立性和安全性。同时，利用PPEA可以实现一定的异步通信，即不需要等到receiver接收完数据后即可着手准备下一次数据传送，成倍的提高了IPC的效率。PPEA算法的原理如图4.2所示：

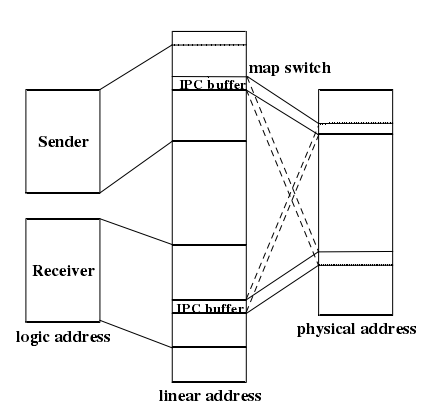


图4.2 物理页面交换算法原理

段基址交换算法（Segment Base address Exchange Algorithm，SBEA）的本质是内核交换sender和receiver的LDT中特定段描述符的段基址，进而使得receiver可以使用某一个选择子指向这个描述符来获取数据。在文章**[3]**是利用32位保护模式下新增的fs段寄存器来指向这个IPC缓冲段。

SBEA相对于PPEA具有两个明显的优势。第一，若进程访问区域超过了段的界限，x86架构为此提供了原生的保护，系统会产生#GP保护性错误；第二，在x86架构中页面映射的最小细粒度是4KB，这样在通信的过程中可能造成内存空间的浪费，譬如只需要传送1KB的数据，却依然需要内核交换4KB大小的页面。而x86架构的段描述符所描述的内存空间大小可以由0到4GB，这意味着SBEA的最小细粒度可以到到1B，有效节约了内存空间。与PPEA相同的是，利用SBEA通信前后sender和receiver的地址空间依然具有独立性，所以该算法能够很好地保证通信双方的信息安全。SBEA算法的原理如图4.3所示：

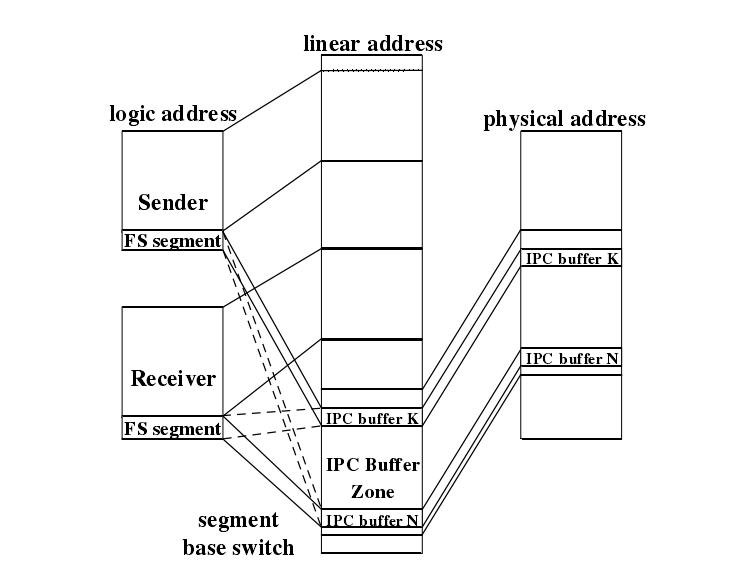


图4.3 段基址交换算法

5 “磋商”型SBEA（N-SBEA）与性能验证实验

5.1 PPEA和SBEA存在的问题

文章**[3]**提出了基于物理页面的快速交换IPC通信算法（PPEA）与基于段机制交换快速IPC算法（SBEA），这两个算法保证了微内核IPC中的高效性和安全性。然而，美中不足的是，在某些应用场景下，这两个算法存在着一些不足。在PPEA中，进行改变的实际上是进程的页表，也就是将通信进程双方各自的某一个IPC buffer物理页（每一个IPC buffer 是一个物理页，大小为4KB）的线性地址进行交换。将sender的IPC buffer的物理地址映射到接收方IPC buffer的线性地址；而将receiver的IPC buffer映射到sender的线性地址中，如此仅仅需要花费修改页表的时间和receiver读取数据的时间，就可以完成进程间的数据交换。对于SBEA，内核管理一个称为IPC buffer 位图的数据结构，每个IPC buffe大小为128B，在每一次进程申请发送数据时就分配一个IPC buffer的线性地址给该进程，同时也给receiver分配一个IPC buffer，并且利用fs段寄存器来指向这一个128B的段，在sender将数据放入缓冲段后，内核对sender和receiver的LDT中指向双方IPC buffer的段基址进行交换，如此当receiver利用该段去访存时，已经可以获得通信数据，且若sender需要继续发送数据，可以不用等到receiver接收完后才能放入数据，因为得到的receiver的IPC buffer是空的（如果只是单向传输的话），由此提高了通信效率。

然而SBEA作为对PPEA的改进，存在如下一个问题。在内核确定好IPC buffer的大小后（这里假定为128B），非常小的通信数据譬如100B、200B等可以在短短几次的段基址交换后完成通信，但如果是有2KB大小的数据需要传输，则需要进行16次段基址交换，而又因为2KB<4KB，所以不可能使用PPEA进行IPC。这样受限于固定的IPC buffer，在最极端的情况下，数据大小为3.99KB，则需要32次段基址交换。如果这类通信在系统中占的比例较大，显而易见，SBEA的性能将大大降低。

5.2 优化策略

由此，本文提出一种称为“磋商”（Negotiate）的方式解决上述问题。在两个进程进行通信前，Sender向内核提出negotiate请求，并发送一个negotiate报文给内核，内核再通知Receiver。若Receiver同意，则发送“同意”（Agree）报文，这样可以充分利用段的大小任意的特点，只进行一次段基址交换就可以完成这个IPC。尽管需要额外花费一次IPC，但这种代价相比与前文所述的最差要进行32次段基址交换而言显然是值得的。

5.3 SBEA与N-SBEA的实现

本论文中基于消息传递机制对SBEA和N-SBEA分别进行了实现。在SourceCode/kernel/process.c中实现一个exchange(PACKAGE\* pPackage)函数，利用它向系统任务进程sys\_task发送一个报文，其中封装了一个PACKAGE\* 变量用于向sys\_task说明所要交换段基址的两个进程，结构体PACKAGE的作用有两个，第一个是在进行“磋商”的过程中请求sys\_task对所要交换的段的界限进行更改，其中声明了整型变量dataSize；第二个是在调用exchange()时将sender和receiver在PCBTable中的索引传给sys\_task，结构体PACKAGE\*如下：

typedef struct package {

int source;

int destination;

int dataSize;

}PACKAGE;

在sys\_task完成段基址交换后，在发送一个“完成”报文，将其中的成员变量reply赋为1，表示所要求的服务已完成。由于实现的是同步消息机制，exchange()中利用一个while循环模拟进程阻塞，等待sys\_task发送回复，再继续运行原来的进程。exchange(PACKAGE\* pPackage)函数和sys\_task中交换段基址的核心代码如下：

PUBLIC int exchange(PACKAGE\* pPackage) {

int result=1;

MESSAGE msg;

resetMessage(&msg);

msg.source = 2;

msg.dest = 1;

msg.type = SEG\_EXC;

msg.pPackage = pPackage;

msg.reply = 0;

result = send\_recv(SEND, 1, &msg);

do{

send\_recv(RECEIVE,1,&msg);

}while (!(msg.reply));

return result;

}

case SEG\_EXC: {

PACKAGE\* pPack = msg.pPackage;

PCB\* pSource = PCBTable + pPack->source;

PCB\* pDest = PCBTable + pPack->destination;

u32 seg\_base = ((pSource->ldt[2]).baseHigh2) << 24 | (pSource->ldt[2]).basehigh1 << 16 | (pSource->ldt[2]).baseLow;

pSource->ldt[2].baseHigh2 = pDest->ldt[2].baseHigh2;

pSource->ldt[2].basehigh1 = pDest->ldt[2].basehigh1;

pSource->ldt[2].baseLow = pDest->ldt[2].baseLow;

pDest->ldt[2].baseHigh2 = (u8)seg\_base >> 24;

pDest->ldt[2].basehigh1 = (u8)seg\_base >> 16;

pDest->ldt[2].baseLow = (u16)seg\_base;

ack = 1;

MESSAGE m1;

resetMessage(&m1);

m1.type = SBEA;

send\_recv(SEND, 3, &m1);

do {

send\_recv(RECEIVE, 3, &m1);

} while (!(m1.type == ECHO));

msg.reply = 1;

send\_recv(SEND, src, &msg);

break;

}

这里规定，每一个进程的LDT的第2个段描述符用于做IPC缓冲段，因此sys\_task所要做的就是将sender和receiver的LDT[2]中的段基址进行互换。

对于N-SBEA，为了实现“磋商”的过程，实现了negotiate (PACKAGE\*pPackge)函数，某一个欲通信进程在利用exchange()请求sys\_task进行段基址交换前先利用negotiate()通知sys\_task改变指定段的段界限，成功后以同样的方式发送“完成”报文给请求进程。Negotiate(PACKAGE\* pPackage)的定义和sys\_task中改变段界限的代码如下：

PUBLIC int negotiate(PACKAGE\* pPackage) {

int result = 1;

MESSAGE msg;

resetMessage(&msg);

msg.type = NEGO;

msg.pPackage = pPackage;

msg.reply = 0;

result=send\_recv(SEND, 1, &msg);

do{

send\_recv(RECEIVE,1,&msg);

}while (!(msg.reply));

return result;

}

case NEGO:{

PACKAGE\* pPack = msg.pPackage;

PCB\* pSource = PCBTable + pPack->source;

PCB\* pDest = PCBTable + pPack->destination;

int seg\_limit = pPack->dataSize-1;

printf("pPack->dataSize=%d B\n", pPack->dataSize);

pSource->ldt[2].limitLow = (u16)seg\_limit;

pSource->ldt[2].limitHigh\_attr2 = (pSource->ldt[2].limitHigh\_attr2 & 0x0F) | ((seg\_limit >> 12) & 0xF0);

pDest->ldt[2].limitLow = (u16)seg\_limit;

pDest->ldt[2].limitHigh\_attr2 = (pSource->ldt[2].limitHigh\_attr2 & 0x0F) | ((seg\_limit >> 12) & 0xF0);

msg.reply = 1;

send\_recv(SEND, src, &msg);

}

5.4 性能验证实验与结果分析

本章节要针对论文中的提及的三种IPC算法进行时间测试。主要的内容是，在单向传输同等的数据量的情况下，得出三个算法的时间，并绘制成折线图便于比较。我们采用的内核编译环境为64位操作系统，版本为CentOS Linux release 7.9.2009(Core)，Linux内核的版本号是3.10.0-1160.el7.x86\_64。C语言编译工具采用的是GNU开源编译工具gcc，版本号为4.8.5，汇编语言编译工具采用的是nasm，版本号为2.10.07。而内核的运行环境我们采用bochs虚拟机软件模拟硬件环境，配置文件已经在2.5节中给出。

为了验证本文所提出N-SBEA算法，本文通过在我们设计的微内核中，采用几种不同的通信策略来进行了性能对比测试实验。该内核中以时间片轮转的方式切换任务，时间片大小为10ms。整个系统在完成基本的初始化后，由内核跳入sys\_task进程，开始TestA和TestB的通信。时间的衡量标准采用通信前后的时钟中断发生次数，在SourceCode/kernel/include/global.h中声明了Ticks变量，在每次时钟中断发生时ticks加一。数据量以B为单位，三个算法分别进了大小为1000\*108B、1500\*108、2000\*108、2500\*108、3000\*108、3500\*108、4000\*108、5000\*108。测试数据都是108B的整数倍是由于128B的MESSAGE结构体中有效数据量为108B。测试结果如图5.3所示，因为纵坐标时间以ms为单位，同时横坐标以B为单位，数字都相对较大，故对横纵坐标都取了log对数。

图5.3

由图5.3可以看出，N-SBEA能够在高频IPC的情况下能够明显优于SBEA和COPY这两种IPC算法，在相同通信量的情况下，N-SBEA相比COPY，性能提升了97.35%左右；相比于SBEA，性能提升了93.71%左右，这个提升对于特定IPC情境的微内核性能的改善是十分巨大的。尽管这次实验是在相对理想的环境下（即两个进程没有I/O行为，只进行单向高频数据传输）进行的，但是即使是在真实的微内核多进程环境下，仅仅花费一次通信的代价就能够最大程度利用段机制的特点，一次性将传送尽可能多的数据，有效减少了IPC的次数。

6全文总结与展望

6.1 全文总结

本论文首先从操作系统的发展历史引出微内核的概念，介绍了微内核架构的特点以及如今微内核架构在业界和学界的受到了广泛的关注，得出了微内核架构是未来操作系统发展的一个大趋势，宏内核或者混合内核尽管占据绝大多数市场，但是从Google发展Zircon和华为发展鸿蒙操作系统可以看出，微内核在未来必然在操作系统的市场上占据一席之地，甚至说全面取代宏内核。论文还分析了当前国内外的一些发展现状，得出了在当前发展环境下对微内核架构的IPC进行研究的必要性。之后，论文对本次微内核实现过程中的相关基础知识和技术进行了介绍，包括x86架构、GCC、Bochs虚拟机等技术，并对当下几种微内核中进程间的通信模型进行了详细的介绍，包括共享存储器系统、消息传递系统、管道通信系统和客户机-服务器系统。在这之后，详细剖析了本论文所实现的微内核，并辅以代码说明具体的实现过程和思路，为后面实现三个IPC算法做下铺垫。

最后，论文分析了复制型IPC和段基址型IPC的缺陷，提出了基于“磋商”型的SBEA算法，这种算法能够有效减少SBEA在特定IPC情境下的IPC次数。然后，本文设计了相关测试实验验证了N-SBEA算法在性能上的优势。

6.2 存在的不足

由于本论文包括了设计一个微内核和进行IPC实验，工作量十分巨大，涵盖内容非常广，所以在微内核实现的过程中，在所难免会存在不少bug和没有考虑到的情况，进而导致实验环境没有办法逼近真实的微内核多进程环境。尤其是两个进程之间进行通信并没有考虑到进行I/O操作的可能，使得在真实环境中，性能优化的效果可能没有在本实验环境下那么好。

6.3 未来展望

本论文以逼近真实IPC环境为目标，对SBEA进行优化从而提出N-SBEA，将继续展开更加深入的研究，下一步工作目标包括：

（1）完善本论文所实现的微内核，增加注入文件管理，内存管理等模块。

（2）将本论文设计的内核在64位硬件环境下进行实现。

（3）继续对N-SBEA进行优化研究，希望能够将其放在其他成熟的微内核环境中进行实验测试。

参考文献

[1] William Stallings. Operating System: Internals and Design Principles(6th edition)[M]. Beijing: Publishing House of Electronics Industry, 2010:36

[2] 汤小丹, 梁红兵, 哲凤屏, 汤子瀛. 计算机操作系统(第4版)[M]. 西安: 西安电子科技大学出版社, 2014: 24-74

[3] 彭星海. 基于x86架构的微内核操作系统的研究与实现[D]. 电子科技大学, 2020

[4] 王宽卿. 微内核进程间通信的研究[D]. 浙江大学, 2010

[5] 中文维基百科. L3微内核[EB/OL]. (2021-2-7) [2021-3-6]. htttps://www.wiki-wiki.top/wiki/L3

[6] 中文维基百科. L4微内核[EB/OL]. (2021-2-26) [2021-3-6]. <https://www.wiki-wiki.top/baike-L4>微内核系列

[7] 冯丽. 基于异构多核环境下微内核操作系统通信机制的研究[D]. 桂林理工大学, 2019

[8] 王琼. UNIX中如何利用管道实现进程间通信[J]. 广东金融电脑, 1999, 11: 40-41

[9] 段莹, 管涛. Linux进程间管道通信的研究[J]. 软件导论, 2012,11(7): 3-5

[10] 周亮, 王震, 王冠. 远程过程调用技术在分布式软件系统中的应用[J]. 航空电子技术, 2020,51(4): 47-52

[10] 高云志. 远程过程调用协议RPC在异构双核通信中的应用[J]. 火控雷达技术, 2013,42(2): 74-78

[11] 于渊. Orange’S:一个操作系统的实现[M]. 北京: 电子工业出版社, 2009: 75

[12] Intel Corporation. INTTEL 80386 PROGRAMMER’S REFERENCE MANUAL 1986[S]. California: 1987

[13] Z. Mi, D. Li, Z. Yang, et al. Skybridge: Fast and secure inter-process communication for microkernel[C]. Proceedings of the Fourteenth EuroSys Conference 2019, 2019, 1-15

[14] 中文维基百科. x86-64[EB/OL]. (2021-2-9) [2021-3-30]. [https://www.wiki-wiki.top/](https://www.wiki-wiki.top/baike-L4)wiki/X86-64

[15] 田宇. 一个64位操作系统的设计与实现[M]. 北京: 人民邮电出版社, 2018: 226

致 谢

附录 内容名称

以下内容可放在附录之内：

(1) 正文内过于冗长的公式推导；

(2) 方便他人阅读所需的辅助性数学工具或表格；

(3) 重复性数据和图表；

(4) 论文使用的主要符号的意义和单位；

(5) 程序说明和程序全文。