# 走向真正的微内核操作系统

MINIX的修正使它在性能上大幅度提升并且通过将设备驱动移向用户态缩小了内核体积。

# 目录

[走向真正的微内核操作系统 1](#_Toc162409610)

[目录 2](#_Toc162409611)

[摘要 5](#_Toc162409612)

[序言 6](#_Toc162409613)

[第一章 绪论 7](#_Toc162409614)

[1.1 操作系统 7](#_Toc162409615)

[1.1.1 基本概念 7](#_Toc162409616)

[1.1.2 操作系统结构 8](#_Toc162409617)

[1.1.3 内核特性 9](#_Toc162409618)

[1.2 微内核操作系统 9](#_Toc162409619)

[1.2.1 微内核应用 10](#_Toc162409620)

[1.2.2 微内核的安全性 11](#_Toc162409621)

[1.2.3 运行效率问题 11](#_Toc162409622)

[1.3 MINIX介绍 11](#_Toc162409623)

[1.4 问题表述 12](#_Toc162409624)

[1.5 研究步骤 13](#_Toc162409625)

[1.6 本文大纲 14](#_Toc162409626)

[第二章 问题分析 15](#_Toc162409627)

[2.1 时间测量 15](#_Toc162409628)

[2.1.1 测试方案 15](#_Toc162409629)

[2.1.2 测试结果和讨论 16](#_Toc162409630)

[2.2 为系统服务设计选项 16](#_Toc162409631)

[2.2.1 系统映像包含什么 16](#_Toc162409632)

[2.2.2 按需求装载系统服务 17](#_Toc162409633)

[2.2.3 本项目采取的方法步骤 17](#_Toc162409634)

[2.3 依赖性分析 17](#_Toc162409635)

[2.3.1 怎样消除依赖性 20](#_Toc162409636)

[2.3.2 功能分类 20](#_Toc162409637)

[2.3.3 消息顺序问题 21](#_Toc162409638)

[第三章 改进内核 23](#_Toc162409639)

[3.1 对用户态设备驱动程序的支持 23](#_Toc162409640)

[3.1.1 为I/O设备增加的系统调用 23](#_Toc162409641)

[3.1.2 通用的虚拟拷贝 24](#_Toc162409642)

[3.1.3 中断处理 25](#_Toc162409643)

[3.1.4 获得系统信息 26](#_Toc162409644)

[3.1.5 对支持功能的支持 28](#_Toc162409645)

[3.2 进程间通信 28](#_Toc162409646)

[3.2.1 会合消息传递（Rendezvous message passing） 28](#_Toc162409647)

[3.2.2 会合进程间通信的实现 29](#_Toc162409648)

[3.2.3 MINIX系统调用的修正 29](#_Toc162409649)

[3.3异步事件的处理 32](#_Toc162409650)

[3.3.1 原始实现 32](#_Toc162409651)

[3.3.2 一个新的通知结构 33](#_Toc162409652)

[3.3.3 异步事件的处理 35](#_Toc162409653)

[3.4 一个新的关闭序列 35](#_Toc162409654)

[3.4.1 原始实现 36](#_Toc162409655)

[3.4.2 关闭MINIX的新设置 37](#_Toc162409656)

[3.4.3 将来的修改 40](#_Toc162409657)

[第四章 新的应用 41](#_Toc162409658)

[4.1 新的消息服务机制 41](#_Toc162409659)

[4.1.1 MINIX2.0.4的调试 41](#_Toc162409660)

[4.1.2 一个分布式的方法 41](#_Toc162409661)

[4.1.3 观察功能码 41](#_Toc162409662)

[4.1.4 惊恐之后的调试 41](#_Toc162409663)

[4.2 看门狗计时器的一般管理 41](#_Toc162409664)

[4.2.1 原始实现 41](#_Toc162409665)

[4.2.2 定时器的一般管理 41](#_Toc162409666)

[4.3 处理无反应硬件 42](#_Toc162409667)

[4.3.1 MINIX2.0.4的异常处理 42](#_Toc162409668)

[4.3.2 察觉超时的新方法 42](#_Toc162409669)

[第五章 内核缩小 43](#_Toc162409670)

[5.1 用户态的PRINTER驱动 43](#_Toc162409671)

[5.1.1 原始PRINTER任务的修改 43](#_Toc162409672)

[5.1.2 用户态的PRINTER驱动的安装 43](#_Toc162409673)

[5.1.3 一般的多级调度 43](#_Toc162409674)

[5.2 用户态MEMORY驱动 43](#_Toc162409675)

[5.2.1 用户态MEMORY驱动的安装 43](#_Toc162409676)

[5.2.2 读实时时钟 43](#_Toc162409677)

[5.2.3 ‘ps’效用的问题 43](#_Toc162409678)

[5.3 用户态的AT\_WINI和FLOPPY驱动 44](#_Toc162409679)

[5.3.1 检测控制器超时 44](#_Toc162409680)

[5.3.2 驱动独立代码的改变 44](#_Toc162409681)

[5.3.3 动态驱动控制器映射 44](#_Toc162409682)

[5.4 用户态TTY驱动 44](#_Toc162409683)

[5.4.1 MINIX关闭代码的重新设计 44](#_Toc162409684)

[5.4.2 调试 44](#_Toc162409685)

[5.4.3 输出诊断 44](#_Toc162409686)

[第六章 相关工作 45](#_Toc162409687)

[6.1 CMU（Control Monitor Unit 监控装置） Mach 45](#_Toc162409688)

[6.1.1 内核特性 45](#_Toc162409689)

[6.1.2 Mach的应用 45](#_Toc162409690)

[6.2 QNX Neutrino RTOS（Real-Time Operating System 实时操作系统） 45](#_Toc162409691)

[6.2.1 系统的体系结构 45](#_Toc162409692)

[6.3 L4微内核API 45](#_Toc162409693)

[6.3.1 L4的实现 45](#_Toc162409694)

[6.3.2 L4Ka::Pistachio 45](#_Toc162409695)

[6.3.3 L4应用的例子 46](#_Toc162409696)

[6.4 与MINIX的比较 46](#_Toc162409697)

[第七章概要和结论 47](#_Toc162409698)

[7.1 贡献 47](#_Toc162409699)

[7.2 回顾 47](#_Toc162409700)

[7.3 将来的工作 47](#_Toc162409701)

[附录A 源代码树的组织 48](#_Toc162409702)

[A1 内核文件总览 48](#_Toc162409703)

[A2 新源代码树的组织 48](#_Toc162409704)

[附录B 怎样应用改变 49](#_Toc162409705)

[B1 增加程序到系统映像中 49](#_Toc162409706)

[B2 增加系统调用 49](#_Toc162409707)

[B3 增加系统库 49](#_Toc162409708)

[附录C MINIX的系统调用 50](#_Toc162409709)

[C1 系统调用实现的组织 50](#_Toc162409710)

[C2 MINIX3.0.0中系统调用总览 50](#_Toc162409711)

# 摘要

操作系统构成了所有用户计算机活动的基础。因此，它应当是可信赖的并且功能齐全的。不幸的是，当今的操作系统，如Windows和Linux都没有正确传达这个思想，因为它们在基础设计时就存在大量的缺陷和bug。它们的单内核结构趋向于功能超载，这些相关功能的模块都运行在特权级。这种结构很容易引进bug，再加上所有相关的风险而打破最小授权原理Principle of Least Authorization (POLA)。例如，一个第三方的设备驱动故障能够很容易地引起系统得严重破坏，并且以完全故意伤害的状态离开。

微内核操作系统有一个完全不同的设计理念，这使得它们不容易受到上述问题的攻击。微内核仅仅提供一个最小集，它们运行在特权级。而扩展的操作系统功能则作为用户空间的服务。通过将操作系统分离为小的独立的模块，系统变得不再那么复杂并且精力充沛，因为更小的模块更易于管理，并且有助于将各种错误分开处理。

本文描述了一个通过开发模块性而创建一个更可靠的操作系统得成果。MINIX被选为这个项目的基本操作系统，因为它已经是一个小而简单同时支持POSIX标准的系统。MINIX的内核表现为一个混合的微内核，因为它包含设备驱动。然而，MINIX的内存管理系统和文件系统已经作为独立的用户态服务实现了。

本文的主要贡献是，MINIX已经完全修正成为一个真正的操作系统。在内核态，增加了系统调用以支持用户态设备驱动，改进了MINIX的进程间通信（IPC）工具，实现了一个新的关闭序列（shutdown sequence）。在用户态，建立了一个新的消息服务机制用来处理调试，另外还创建了一个库用来维持一个看门狗计时器列表。通过将打印机、存储器、软盘驱动器等各种终端设备转移到用户态设备驱动，这些更改使缩小内核体积成为可能。

# 序言

# 第一章 绪论

操作系统是最底层的软件，它控制计算机的硬件并且给上层的应用程序提供基础。操作系统构成了所有用户计算机活动的基础。因此，它应当是可信赖的并且功能齐全的。不幸的是，当今的操作系统，如Windows和Linux都没有正确传达这个思想。大多数用户已经习惯于频繁的操作系统崩溃，并为病毒（digital pests）而烦恼。

究其原因，当今的操作系统在基础设计时就存在大量的缺陷和bug。它们的设计表现为一个巨大的单一模块程序，该程序处于机器的完全控制下。一般的设计指导方针，例如模块化及最小授权原理（principle of least authorization）等，被阻碍而没有得到有力执行。例如，第三方的设备驱动通常是内核的主要组成部分并且拥有全部的特权，有了这些特权它们可以使整个系统崩溃。

另一个主要的问题是bug的不可避免的存在。软件统计表明，每1000行代码就存在1到20个bug。对操作系统这种比普通软件更复杂的软件来说，这个数字甚至可能会更糟糕。当今大部分广泛使用的操作系统有上百万行代码，因而很容易有上千个bug。例如Windows XP有5000万行代码，在最糟糕的情况下会有100万个bug。

在这个项目中，我们通过模块性开发努力创建一个可信赖的操作系统。通过将操作系统分离为小的独立的模块，系统变得不再那么复杂并且精力充沛，因为更小的模块更易于管理，并且有助于将各种错误分开处理。MINIX被选为这个项目的基本操作系统，因为它已经是一个小而简单同时支持POSIX标准的系统。主要的目标是将MINIX的设备驱动转换成独立的程序，这样MINIX便可成为一个真正的微内核操作系统。

本章的剩余部分结构如下。1.1节对操作系统原理做了一个简要介绍，1.2节详细讨论了微内核操作系统，1.3节介绍了MINIX，1.4节提供了一个准确的问题描述，1.5节提出了本项目采取的途径。最后，1.6节大体描述了本章剩余章节的内容。

## 1.1 操作系统

这一节介绍一般性的操作系统原理，下一子节介绍一些重要的操作系统概念，1.1.2子节将不同的操作系统结构分类，子节1.1.3对比较单模块操作系统合微内核操作系统的基本性能。

### 1.1.1 基本概念

一般来说，关于操作系统由两种观点。它们可以被看作一个资源管理器或是一个虚拟机。资源管理器负责提供对计算机硬件的访问并且安全地同时处理所有需求。例如，资源可以包括硬盘、CD-ROM、打印机和图像存储器，资源也可以被多个应用程序共享。

扩展计算机意味着使用更高级的功能提高硬件性能。例如，传统的操作系统可能提供文件系统和窗口功能。扩展计算机为应用程序定义接口。与资源管理相比，这个接口可以在不同的硬件体系结构间共享。

内核是操作系统运行在最高特权级的那一部份，被称为内核态或特权模式。内核态涉及了一个CPU标志，用以表明正在运行的进程是否被允许执行所有可能的指令。运行在用户态的程序没有设置这个标志位，因此不允许执行某些指令。

操作系统接口由系统调用所定义，程序员可以使用这些系统调用。系统调用使用一个特殊的机器指令，叫做一个陷阱，它会引起处理器从用户态转换成内核态并且分配内核的系统调用处理程序。用户进程以这种方式要求系统服务执行如访问硬件等受限制的行为。

通过进程间通信，进程可以互相交流，这通常由一个系统调用实现。例如，IPC可以通过传递一个跟随着上下文切换的请求消息来实现。上下文切换意味着停止正在运行的进程并且保存它的状态，这样另一个进程可以恢复并重新运行。接下来，存储进程状态之后发出一个特别的机器指令。



图1.1 操作系统的分级，单一模块内核(a)通常是无组织的，微内核(b)通常由一个分层的结构，Exokernel(c)是一个垂直结构的操作系统

操作系统得一个关键问题是确保用户应用程序不能与内核接触，并且它们之间也不可以接触。这是通过在物理上彼此受保护的地址空间中封装进程实现的。内存管理单元（MMU）通过验证每个试图访问内存的行为来加强这种保护。非法的访问将导致异常，该异常会被内核中的一个异常处理程序捕捉并采取合适的行为。

### 1.1.2 操作系统结构

操作系统可以基于它们的内核设计来分类。资源管理器和扩展计算机两种操作系统观点的区别对不同结构的分类是相关的。主要有三个主要的分类：

**单一模块内核结构：**

单一模块内核结构提供了丰富和强大的底层硬件的抽象。所有的操作系统服务被编译为一个单独的程序运行于内核态，而应用程序运行于用户态并且可以从内核申请系统服务。因此内核既是资源管理器，也是一个扩展计算机。从一个高级的角度看，单一模块的内核是无组织的。这在图1.1(a)中已有描述。Windows、BSD UNIX和Linux都是单一模块内核结构的操作系统。

**微内核及混合内核的结构：**

微内核的特点是它是内核抽象的最小集，但并不需要在内核尺寸上达到最小。微内核提供了一个简单硬件抽象的小集合并且使用称之为服务的应用提供更多的功能。只用操作系统的一个最小部分运行在内核态，而所有的应用运行在用户态。与单一内核结构比较，微内核的主要功能是作为一个资源管理器。微内核操作系统与客户端服务有一个松散的层次界结构，如图1.1(b)所示。L4是一个真正的微内核的例子。

混合内核结构与微内核相似，但是由于运行方面的原因将一些额外的服务增加到内核态来运行。例如MINUX和Mach就属于这类内核结构，因为它们的设备驱动程序运行在内核态。QNX也是一个混合的微内核结构，因为它的进程管理系统实内核的一部份。

**虚拟机和exokernel：**

虚拟机和exokernel不像其他操作系统那样提供一个硬件抽象层；相反地，它们分别拷贝或分割硬件资源，这样多个操作系统可以互相接着运行，给人们一种拥有各自的计算机的假象。一个虚拟机监控器或exokernel运行在内核态并且负责资源的保护以及多路硬件请求，而每个操作系统运行在用户态，彼此完全独立。虚拟机和exokernel都被认为是垂直结构的操作系统。一个接一个运行的操作系统或者有一个单一模块内核，或者是微内核结构，如图1.1(c)所示。VMware和MIT exokernel分别是虚拟机和exokernel。

因为虚拟机和exokernel仅仅是一种提供和操作系统接口的手段，就不进一步讨论它们了。本文感兴趣的是单一模块内核与微内核的区别。

### 1.1.3 内核特性

操作系统特性的总览见图1.2。我们可以看出，拥有单一模块内核结构的操作系统通常情况有很好的运行效率，而微内核结构的操作系统在其他所有方面都做得更好。正是模块性的区别直接滋生出这些更多的区别。



图1.2 单一模块内核和微内核结构的操作系统特性的比较，再子节1.2.3中，我们会讨论，基于微内核结构的操作系统的执行效率并不是那么坏

模块性正是使微内核如此强大的关键特性。与单一模块内核的系统相比，微内核操作系统通常作为共同运转的服务的集合来实现，每个服务都有它自己的任务。这就对所有相关服务做了一个清晰的责任分离。

微内核操作系统中的特性的例子如下。微内核是灵活的并且很容易扩展，因为替换服务或是问题解决策略成为可能。它们更容易维护，因为小的组件不再复杂而更易于管理。责任的分离有益于安全和操作系统的精力充沛，因为错误被孤立起来，故障组件可以随时被替换。有关安全方面的细节可参见1.2.2。

运行效率是一个人们长期反对微内核的特性。单一模块内核系统通常具有良好的运行效率，因为所有的服务在内核中运行，可以直接访问彼此。相反，微内核操作系统为了使各种系统服务彼此协作需要额外的通信。正是由于这个原因，人们总是认为微内核系统运行效率低下，然而事实并不是这样的。子节1.2.3详细地讨论了这个问题。

微内核的一个有趣的特性是它使得保持一个UNIX环境同时试验新应用成为可能。它们提供向后兼容同时转换到新的计算环境，在子节1.2.1种将进一步讨论。

虽然潜在的利益是巨大的，然而微内核操作系统成功的例子很难找到。QNX也许实现了大多数承诺，但不幸的是，它是一个封闭的商业系统。本项目可能帮助MINIX占领开源微内核操作系统的一席之地，并且将来的工作也将基于这个项目发展下去。

## 1.2 微内核操作系统

前一节介绍了操作系统得一般原理，这一节集中集中讨论微内核操作系统。子节1.2.1论述了三种在微内核之上 构建应用的不同方法。子节1.2.2和1.2.3分别讨论了两个重要的操作系统特性——安全和运行效率。



图1.3 基于运行在用户态程序的结构对微内核应用分类。不同的结构为，(a)是一个有专门组件的单服务操作系统，(b)是一个多服务的系统，(c)是一个专用系统

### 1.2.1 微内核应用

在子节1.1.2中，我们大体知道，微内核和混合内核相对单一模块内核提供了一个简化的抽象集合。因此，微内核需要依靠用户态服务为运行在操作系统治上的应用程序提供额外的功能。这种功能可以用几种不同的方式实现。

**单服务操作系统：**

在单服务操作系统中，微内核运行整个单模块操作系统，就像是运行一个普通的用户程序，如图1.3(a)所示。这种组织不会改变单模块操作系统的任何特性，这意味着它仍然是一个单一模块内核的操作系统。这种组织的主要优势是允许保持一个UNIX环境而同时对微内核试验。面向单模块操作系统的遗留应用可以和新应用共存。遗留的应用和时势或安全组件的联合构成了向新计算环境平滑的过渡。

有很多这种方法的例子存在。Mach是第一个运行多个操作系统特征的微内核，包括BSD UNIX和OSF/1。另一个例子是L4Linux，它运行在L4微内核之上。PERSEUS项目努力运行安全数字签名专门的组件。

**多服务操作系统：**

在多服务操作系统中，操作系统环境由一系列协同操作的服务组成，如图1.3(b)所示。正如子节1.1.3中所描述的那样，增加的模块性带来大量益处，如使系统精力充沛，提高可维护性和灵活性。依靠多服务环境提供的功能，如果与一个仿真库链接，那么遗留应用仍是可用的。可以通过写一个新的系统服务来简单实现新的应用。

有几个多服务操作系统得例子。例如，GNU Hurd就是一个运行在修改过的Mach微内核之上的多服务环境。SawMill Linux是一个运行在L4微内核之上的多服务环境。1.3节将讨论这种类型操作系统得另一个例子，MINIX。

**专用系统：**

另一种组织方式是使用一个专门的应用，该应用直接运行于微内核之上，如图1.3(c)。这种系统得变化很多，在移动电话和嵌入式设备这种简化了计算机功能的设备上特别有用。

专用系统得一个例子是Java虚拟机（JVM），它直接运行在微内核之上。与操作系统控制JVM这种传统的方法相比，它为Java应用实现了一个更加安全的环境，因为主机不再需要被信任。

### 1.2.2 微内核的安全性

如1.1.3节中描述的那样，微内核有着良好的安全性。这对于所有系统服务都封装在地址空间的多服务操作系统来说更是如此。服务彼此间被内存管理单元（MMU）物理性地保护起来。如果一个进程试图非法访问另一个进程的内存，将会被MMU检测到并抛出一个异常。内核捕捉到异常并采取它认为必要的活动。

虽然一个服务并不能直接破坏其他服务，但是相关的服务可能会被间接地影响。如果依赖于服务B的服务A在执行一个任务，服务A可能会受到服务B中的一个故障或者服务B的一个恶意行为的影响。一个与之相关的重要概念是信赖计算基础（trusted computing base，TCB），也就是说，正确功能是一个安全预处理的组件的最小集。对于微内核而言，TCB可以非常小。例如，在一个专有系统里，一个专有应用的TCB仅仅是微内核及底层硬件。

一个重要的不容易解决的安全问题是硬件控制的组件有能力破坏整个系统。例如，可以使用直接存储访问（direct memory access，DMA）的设备驱动程序可能会通过提供给DMA控制器一个无效地址来破坏任意进程的内存。一些芯片集有一个I/O MMU，通过映射一个PCI地址空间到物理内存得一个已知区域来提供保护。目前还没有可行的普遍解决办法。

### 1.2.3 运行效率问题

通常认为微内核操作系统具有一个与生俱来的运行效率问题，因为多个进程必须通过相互协作来执行一个任务。所谓的运行效率降低的原因是额外的进程间通信，因此需要用户进程、系统服务和内核之间进行额外的上下文切换。此外，互相协作运行的服务之间的数据拷贝也引起了额外的负担。

第一个微内核系统的确有效率的问题。例如，Mach上的BSD UNIX的运行效率比正常版本的BSD UNIX低50%。然而，现在的微内核已经证明可以实现高的运行效率。例如，L4上的L4Linux仅仅损失了2%到4%的运行效率。

运行效率问题变成了一个实现问题。L4中采取的方式是定义一个灵活的微内核接口，但是为了实现这个功能而对硬件性能做了最优化的使用。这样每个平台需要一个不同的内核实现，但是与L4的内核API紧密相关的应用可以重新编译和运行。

另一个重要的问题是微内核必须很小，L4的运行效率低下表明缺少缓存比IPC（进程间通信）本身更重要。从用户态进程到内核态的上下文切换仅仅需要改变CPU少数位，切换到内核态并存储内核的状态。然而，地址空间的切换可能引起缓存空间的严重不足。如果使用了虚拟内存情况会更加糟糕，因为转换后援缓冲器（TLB）的入口由于地址空间切换已经无效。

最后，软件应该为微内核而明确设计。微内核与单一模块内核不同，为了达到好的运行效率它需要不同的设计。例如，SawMill Linux的研究便是针对设计一个消减进程间通信（IPC）和防止不必要的数据拷贝的多服务协议。因此基于微内核的操作系统得运行效率不能仅仅由进程间通信的代价来衡量。

## 1.3 MINIX介绍

本项目集中讨论为内核操作系统，并特别针对MINIX。MINIX是一个免费的基于微内核的操作系统，它的源代码完全开放，大部分程序用C语言编写。Andrew S. Tanenbaum于1987年写了MINIX并作为教学使用，它容易学习并且易于维护。有一本书解释了它的详细设计。

MINIX的设计表现为一个基于混合内核模式的多服务的操作系统，它的结构如图1.4所示。主要的组件，例如文件系统（FS）和内存管理（MM）被设计为运行在用户态的独立服务。我们说MINIX是一个混合内核，是因为它的设备驱动程序被编译为内核的一部分。



图1.4 MINIX 2.0.4表现为一个基于混合内核模式的多服务的操作系统，所有的设备驱动程序是内核的一部分，而重要的系统功能由用户态的服务实现，例如文件系统（FS）、内存管理（MM）和网络服务（INET）

因为设备驱动程序是内核的一部分，它们运行在更高的特权级。这已为MINIX的设备驱动程序是完全值得信赖的，然而实际上不应当是这样。因此，本项目的目标之一就是将设备驱动程序转换成独立的用户态服务，这样，MINIX将变成一个基于真正微内核的多服务操作系统。1.4节中将给出该问题的精确表述。

自MINIX的最初版本以来，源代码树逐渐发展，但是它的内核相对比较稳定。符合POSIX标准的第二版MINIX与1996年完成。

本项目基于2003年11月发布的MINIX2.0.4，它将作为本项目的一个中间过渡。MINIX2.0.4的内核大概包含2万行代码，包括组件。在http://www.cs.vu.nl/pub/minix/可以获得MINIX的其它版本。

附录A1提供了MINIX2.0.4内核所有文件的总览。内核共有77个文件，包括驱动程序在内共878KB大小，微内核部分的代码只有220KB。列表显示只有一小部分（25%）的源代码属于MINIX的微内核，而源代码的大部分（75%）都是由设备驱动程序组成。因而将设备驱动从内核中移出将会导致运行在最高特权级代码的大量减少。

## 1.4 问题表述

本项目的目标是修正MINIX2.0.4位一个更可靠的操作系统。这通过增加它的内核来实现，使它变得更小并且更强壮，因而问题表述是双重的：

**内核减小：**

MINIX的内核可以通过转移设备驱动程序来实现，使驱动程序由运行在内核态转变成运行在用户态。尽管MINIX的设备驱动程序被设计为独立调度的进程，或称之为人物，然而它们是被编译进内核的。这意味着它们运行在最高的特权级，伴随着所有相关的危险。通过将驱动移出内核，使它们作为普通的用户态进程，MINIX将会变得更安全，因为设备驱动程序和内核将被彼此物理分离保护。

**增强内核质量：**

这一步关心的是提升操作系统的总体质量。MINIX的内核包含一些并没有严格要求进入内核的功能，然而，这些代码却访问了内核中的重要数据结构并可能会危及系统。所有这样的代码应该移出内核，例如可以通过建立额外的用户态服务。此外，通过重新设计代码或增加新的安全特性能够改进部分内核代码。

本项目预期的结果实构建一个真正微内核的多服务操作系统。MINIX的新结构如图1.5所示。由于更改结构变化很大，版本号将由MINIX 2.0.4提升到MINIX 3.0.0。



图1.5 本项目的目标之一是将设备驱动程序移出内核，这样MINIX将成为一个真正的微内核多服务操作系统

## 1.5 研究步骤

**熟悉MINIX操作系统：**

本项目从彻底研究MINIX 2.0.0的相关书籍开始，特别注意研究了涉及进程和I/O设备实现的章节。之后对MINIX2.0.4的源代码树进行了研究并对用户态I/O进行了一些原始试验。

**时间测量：**

MINIX的重建引入了一些通信机制，因为用户态设备驱动和内核间必须合作运行和以前一样的任务。在两个不同的机器上引入时间测量机制，用以评估运行效率的损失。2.1节中展示测试过程和结果。

**设计选项的比较：**

MINIX源代码研究发现用户态系统服务可以用两种方式启动。大部分服务被静态地配置到系统映像中，但同时存在一个动态的方法。2.2节将讨论这些可能性的比较。

**可行性研究：**

这一步包含将打印机驱动移出内核，这次成功的转换为本项目剩余部分的完成建立了自信。5.1节中将阐述可行性研究的结果和遇到的问题。

**依赖性分析：**

通过独立编译所有内核任务并搜集所有缺少的符号创建一个依赖性矩阵。分析消除依赖性的不同房法并创建一个按功能的分类，用来找到一种相关依赖性的通用解决办法，用来代替每个设备驱动的特别解决办法。

**设备驱动程序的转移：**

允许有用户态设备驱动的思想是用在内核中执行特权操作的系统调用支持它们。增加许多新的系统到内核中，并且为了清除所有依赖性重新设计了部分MINIX代码。一旦清楚了所有的依赖性，一些设备驱动程序便可以移出内核。第三章到第五章描述了这部分工作。

**改进代码质量：**

消除依赖后，分析更多的MINIX代码。表明还有一些缺点有待改进。因而尽可能对代码做一个整体的改进。

## 1.6 本文大纲

本文的组织结构如下。下一章提供了一个问题分析并简要讨论了不同的设计选项。问题分析主要集中在从内核中移出设备驱动程序过程中扮演重要角色的依赖性问题。

第三章给出了一个MINIX内核改进的总体方案。改进包括支持用户态设备驱动程序的各种新的系统调用、MINIX的进程间通信机制以及MINIX的shutdown序列的完全修正。

第四章论述了各种新应用和内核改进的有效性。新的应用包括获取系统消息的新服务、管理看门狗计时器的一个简单且通用的方法以及处理无反应硬件的新方法。

第五章讨论内核是怎样大幅度缩小体积的。这里主要关心的是将混乱的内核态设备驱动程序转换成为独立的用户态程序。讨论覆盖了PRINTER（打印机）、MEMORY（存储器）、AT WINI、FLOPPY（软盘驱动器）和TTY的设备驱动程序。

第六章调查了微内核操作系统方面的相关工作。讨论了三个典型的微内核系统Mach、QNX和L4，以及它们的一些应用。Mach在历史观点上很有趣，QNX Neutrino是一个成功的商业嵌入式微内核操作系统，L4的特点是体积小并且运行效率高。

最后，第七章做了一个总结，提出了本文的主要贡献，回顾完成了那些工作并得出结论，最后描述了将来工作的可能方向。

最后，附录中展示了文中所没有细节。 附录A提供了MINIX内核的文件列表并描述了新的源代码树，附录B说明了如何改变，附录C列出了MINIX的所有系统调用。

# 第二章 问题分析

这一章展示的是问题分析过程，这是在实际编程开始前所要做的工作。2.1节展示了时间测量用于获得用户态设备驱动程序所带来的影响。接下来2.2节讨论了加载系统服务的不同设计选项。最后，3.3节提供了一个详细的分析和依赖性的分类，在将内核态任务转换成用户态设备驱动时，这些依赖性必须消除。

正如1.5节中所描述的那样，打印机驱动到用户态转换将作为本项目的可行性研究，也可以看作是问题分析的一部分，结果将在第五章中讨论，与其他用户态设备驱动一起。

## 2.1 时间测量

将内核态任务转换为独立的用户态服务需要额外的通信机制，因为服务必须和内核协同工作（也可能是和其它服务协同工作）。例如，设备驱动程序将不再有特权直接执行I/O设备。因此，它们必须要求内核读或写指定的设备记录。

因而，MINIX的重新构造需要额外的上下文切换机制并带来一些执行效率上的损失。因此为了获得执行效率损失的情况，我们需要用一个典型的请求应答序列来测量时间。

### 2.1.1 测试方案

测试在内核中进行，因为我们需要将时钟计数精确到微妙。在测试期间当一个非法请求消息从CLOCK任务发送到FS服务时，FS服务立即回应一个错误消息。在发送消息前收到回应消息后读时钟计数器。测试装置如图2.1所示。



图2.1 对一个典型的请求应答序列测量时间的测试方案。该时间测量过程包含两次消息拷贝、两次上下文切换和判断消息的最小逻辑单位

测量给出了党进程间必须写作执行某个任务时候所损耗的时间情况。该测试方案包含两次消息拷贝、两次上下文切换和判断消息的最小逻辑单位，因为错误的非法消息会立即返回，所以在服务上的实际工作消耗的时间可以忽略不计。这意味着我们测量的是一个典型的请求应答序列的净消耗时间。

这个测试的实现需要对CLOCK任务做一个小小的改变，在src/kernel/clock.c中需要定义一个新的函数用来将8253A计时器精确到微秒。此外，在CLOCK任务中需要实现一个临时的系统调用，用来初始化测试序列。对文件系统服务无需做任何改变。

### 2.1.2 测试结果和讨论

测试使用不同CPU速度的两个Intel计算机，每台机器上进行30次测试，结果如图2.2所示。结果表明额外的请求应答仅需要很少的时间，并且依赖于CPU的速度。因为测试的机器相当旧，如果是在现在的计算机上测试时间会更少。



图2.2 如图2.1中的测试方案的运行结果，在每个机器上进行30次试验，所有的结果都是以微妙为单位。

不幸的是，单独的时间测量并不能提供评估精确代价的足够信息。实际情况下必须考虑到频繁的请求应答，但是这个却很难实现确定。只有一个或者更多设备驱动程序移出内核时这种测试才能做的精确。此外，测试结果很可能依赖于驱动类型。

为了对运行效率的损失给出一个粗略的估计，我们设想一个典型的请求应答序列在一个现代计算机上需要5微秒，用户态的设备驱动程序每秒钟会发出1000个这样的额外请求。这意味着运行效率每秒钟将损失5毫秒，也就意味其对CPU的影响只有0.5%。因此我们说，与从用户态驱动获得的好处相比，这一点运行效率的损失是合理的。

虽然在这里我们测定的时间是一个固定值，但是在实际中请求应答序列的次数可能会受到各种优化的影响。此外，从一个进程到另一个进程的数据拷贝对某类设备驱动而言可能是一个更大的问题。因此，将来的工作将集中于用户态设备驱动执行效率的更为详细的分析，这不在本项目的讨论范围之内。

## 2.2 为系统服务设计选项

MINIX中的系统服务有两种加载方式，标准的方法是将它们静态地加入系统映像中，但也可以用动态的方法。下面将讨论这两种加载方式的比较，并简要讨论本项目所采取的方法。

### 2.2.1 系统映像包含什么

MINIX 2.0.4中启动系统服务的标准方法是将它们包含在根映像中，根映像由程序‘installboot’引入。系统初始化期间，根映像中的所有服务会自动启动。当前，根映像包含内核、内存管理、文件系统服务和INIT进程。由于程序加载的方式，当前映像中还有空间可以加入12个其他程序。

在系统映像中加入其他程序需要更新内核数据结构，该数据结构用于初始化进程表。例如，内核必须清楚有多少程序加入到系统映像中，并且每个程序要有一个名字。附录B中给出了需要采取的精确步骤。

因为新的系统服务影响内核的源代码，因而不易应用到运行系统中。为了做出改变用户必须获得更新的源代码，以及所有补丁并重新编译内核。这需要用到专门的技术且只有系统管理员才能实施。嵌入式系统和单机系统可能不需要更新，但在变化性较大的环境下可能会有问题。

### 2.2.2 按需求装载系统服务

另一个加载系统服务的方式是在操作系统加载之后再动态地加载系统服务。在MINIX 2.0.4中系统服务作为特殊的进程，因此与加载普通用户程序相比，加载系统服务需要一些额外的步骤。系统调用SYS SVRCTL允许一个有特权的进程在初始化期间转换成一个系统进程。下面会描述载入INET网络服务的步骤。

动态加载服务有如下几个好处。首先，它使系统服务的开发和调试简单化，因为它不需要修改内核，也不需要重新编译，或为了安装一个新的服务重启整个系统。另一个好处是易于更新系统服务，特权用户很容易启动系统服务，如果设置了UID位，任何用户都可以启动服务。

另一个优势是对启动服务的数量没有实际的限制。通过增加头文件<minix/config.h>中变量NR PROCS的值就可以预留额外的资源，即进程表的空间，来运行额外的服务。相反，上面所描述的静态方法需要修改底层的启动代码。

不幸的是，MINIX 2.0.4中的系统调用SYS SVRCTL还有一些缺陷，从而很难发挥动态加载系统服务的完全优势。例如，该调用没有队终止系统服务并释放资源提供明确的方法。

**INET开玩笑**

与文件系统和内存管理服务相比，网络服务INET是单独分配的，而没有包含在系统映像中。因而内核不致到 INET是一个特殊的进程。INET就像其它用户进程一样启动，但是它在初始化期间使用系统调用SYS SVRCTL而成为一个系统服务。在这个过程中有三个步骤：

1.在内存管理登记。内存管理模块通过释放一个等待的父进程并剥夺它的所有子进程的权利假装退出，做完这些之后，进程ID和组被改变，从而成为一个服务。

2.在内核的SYSTEM任务中登记。SYSTEM任务会更新进程的类型和优先级。完成之后，INET将不再是一个用户进程，而是一个真正的服务。

3.最后一步是为了管理一个设备，在文件系统登记。文件系统维护一个设备和处理设备的进程的映射表，INET要求管理设备/dev/ipdevice。一旦加入了新的映射，INET就和其他设备驱动一样了。

### 2.2.3 本项目采取的方法步骤

在本项目中，所有的系统服务，如用户态设备驱动，都包含在根映像中。选择如此静态配置是为了简单化和时间限制。

然而，首选的方法仍是动态启动和停止系统服务。因此，动态控制系统服务的适当接口的设计是将来工作中的一部份。

## 2.3 依赖性分析

为了清楚将设备驱动程序移出内核需要做什么，我们得分析它们之间的依赖性。依赖意味着一个符号，即一个变量或函数，不再被一个独立编译的设备驱动程序所直接引用。大多数情况下，这意味着设备驱动程序要依靠内核功能来做这个工作，但是也有其它种类的依赖。这将在2.3.2节中讨论。

通过拷贝一个给定任务的所有文件到一个单独的目录，独立编译该任务，然后通过检查编译结果和缺失符号的链接错误可以发现该任务的依赖性。所有任务采取这种方法分析后可以获得一个它们彼此依赖的列表。

图2.3展示了结果的依赖性矩阵。该矩阵用来对依赖性分类，通过对不同内核依赖性进行分类，便很容易找到解决办法从而实现移植，这会在2.3.2节中讨论。

**未被发现的依赖性**

有一种类型的依赖性用这种方法找不到，也没有在依赖性矩阵中列出，这就是I/O设备相关的依赖性。因为I/O设备函数标准系统库的一部份，已经与设备驱动动态链接起来了。然而I/O设备需要特权从而不能被用户态的驱动所使用。任何读写寄存器的操作将直接导致CUP异常。



图2.3 基于内核任务独立编译后缺失符号的依赖性矩阵，这些符号或者是一个函数（F），或者是一个变量（V）。标记的驱动程序移入用户态，最后两个任务仍然作为内核的一部份

### 2.3.1 怎样消除依赖性

消除不同类的依赖性要用不同的方法。下面分类描述了我们应该怎样消除依赖性，共有五类依赖性问题需要解决：

**驱动程序对内核的依赖：**

所有的设备驱动都依赖于内核功能。例如，大多数驱动程序都要使用一个内核函数实现对任意物理地址的块拷贝。然而用户态进程不允许这样做。这种典型的依赖性可以通过一个系统调用来解决，系统调用请求内核代替用户态进程执行任务。可以在系统库中实现新的系统调用

**驱动程序对驱动程序的依赖：**

驱动程序可能彼此互相依赖。例如，所有的系统服务都依赖TTY驱动程序输出诊断信息。这种依赖型可以通过定义新的消息类型来代替，这种新的消息类型允许驱动程序间互相通信。类似于MINIX的系统调用，这些进程间的调用也可以实现为一个独立的库。

**内核对驱动程序的依赖：**

内核也可能依赖于设备驱动程序。例如，MINIX关闭时所有的设备驱动程序都会停止于内核的一个函数调用。此外，中断处理程序必须是内核的一部份，但是它们要使用设备驱动程序的数据结构。解决这种依赖性的方法类似于上面所讨论的，就是说，内核发出信息通知事件驱动作进一步的处理。然而，我们会在2.3.3节中讨论，发送消息的具体顺序可能会引发一些问题。

**糟糕的设计：**

还有几种依赖性是由糟糕的设计引起的，这些是不必要的依赖，因为某些符号被全局性声明，而它们只在局部使用。这主要涉及到TTY任务的变量。例如，当一个TTY警报离开的时候，CLOCK任务在TTY驱动上直接设置一个事件位。仅仅将声明改为局部就可以解决这种问题。

**False positives.**

最后，在依赖性矩阵中显示的某些符号并不是真正的依赖性问题。这些符号出现在表中是因为SYSTEM和CLOCK也是独立编译的，目的为了找出内核对驱动程序的依赖性问题。当然，这些任务仍然在内核中，向驱动程度提供底层服务，所以这些依赖性可以忽略。False positives应从图2.3 的依赖性矩阵中排除。

### 2.3.2 功能分类

分析图2.3展示的依赖性矩阵找出功能相关的依赖性。通过对符号分组可以形成几个功能类。功能分类有助于找到依赖性的全部分类的一般方法，而不是对每个依赖性做特殊处理。

下面给出了依赖性的粗略分类和消除依赖性的方法。不幸的是，有些依赖性不容易分类，仍然需要特殊的解决办法。此外，不同的分类也不总是彼此独立处理的。

**Handling assertions and panics类：**

这种问题将被局部处理。如果一个服务仅仅破坏了该服务本身，那么它应当被终止。大多数情况下将，不会影响到系统的其它部分不会受到影响，并能继续工作。

**数据拷贝类：**

MINIX 2.0.4中大部分数据拷贝使用了物理寻址。拷贝应该由SYSTEM任务而不是请求的进程来完成。对于这类依赖性，我们通过定义一个新的通用的拷贝函数来解决，该函数使用虚编址，而不是物理寻址。

**进程表信息类：**

虽然咋一看进程表引起了相当多的依赖性，然而事实上这并不是一个真正的问题。大部分引用是用来计算拷贝数据的实际物理地址，而当引用了虚拷贝之后，这就不再需要了。其他引用的情况下以项讨论。

**获取系统信息类：**

因为驱动程序不再访问重要的系统信息，例如内核的环境变量，因此它们应该能够请求一个重要信息的拷贝。对于这种依赖性问题，创建一个新的系统调用。

**使用CLOCK功能类：**

不像警报定时和延迟那样直接调用CLOCK任务，这种依赖性问题可以通过发送一个请求消息来解决。创建类似于系统库的时钟库函数。

**调试类：**

建立一个单独的消息服务处理调用，这在MINIX 2.0.4中是在内核中进行的。为了获得内核数据结构，还要建立一个请求系统信息的新的系统调用。

**中断处理类：**

中断处理需要很多特权，因此应该在内核中留有通用的中断处理程序，并增加一个新的系统调用用来启用和禁止中断。此外，驱动程序应该设置一个中断策略，用来被内核中通用的终端处理程序调用执行。

**关闭MINIX类：**

关闭是一个全球性的事件，它是在TTY任务中处理的。因此，代码应该被移到一个中心位置，例如也包含MINIX主程序的的文件。这将由一个通知消息来处理，而不是直接调用其它驱动程序。

**变量声明类：**

在MINIX 2.0.4中，很多TTY变量被全局性地声明，实际上这是不必要的。这个设计问题可以如下更正，将这些变量移入一个不同的头文件中，并改变分配给它们的存储方式。

**I/O设备类：**

上面提到过，I/O设备形成了一类依赖性，依赖性矩阵中并没有列出来。因为I/O设备需要特权，因此要增加几个系统调用来让内核代替用户态的驱动程序执行输入输出功能。

### 2.3.3 消息顺序问题

2.3.1节讨论了各种不同的依赖性，如果所有的依赖性都是组织管理严密的，例如驱动对内核的依赖，通过增加额外的系统调用，一个通用可行的方法将消除这个依赖性

然而不幸的是，内核对驱动的依赖也同时存在，更严重的是循环依赖的存在。一个设备驱动程序可能不仅依赖于内核，但是内核可能同时依赖这个设备驱动程序，如果它们互相发送消息，就有可能导致死锁。

MINIX的原始设置中有一个有偏见的消息顺序，用于普通消息的交流，这个顺序是严格的。用户进程可以向内存管理和文件系统服务发送消息，内核的驱动程序可以向CLOCK任务发送消息，如图2.4所示。为了防止死锁，不符合顺序的消息不能以通常方式发送，除非该消息涉及一个对严格请求的回应。因此，异步的事件，如中断和时钟都是用特殊方法来处理的。



图2.4 MINIX 2.0.4中偏心的消息顺序，为了防止死锁所有的请求消息都必须遵从这个顺序。也允许不符合顺序的请求，但在该图中没有显示

MINIX 2.0.4元代码的分析显示，处理异步事件的方法存在某种缺陷，不适合消除所有不符合顺序的依赖性。因此，一个更通用的解决办法是通知系统事件涉及的用户态进程。简而言之，一个新的通知结构将详细检查，在发送消息前，进程是否已经准备好接受一个不符合顺序的消息。如果接受进程没有在等待消息，通知将被延迟。各种异步事件的处理将在3.3节中讨论。

# 第三章 改进内核

这一章中提出了对MINIX内核的各种改进措施。3.1节介绍了为支持用户态设备驱动程序而增加的最重要的系统调用。3.2节讨论MINIX进程间通信机制的改进，包括系统调用处理程序的重写。3.3节讨论了一个围绕进程间通信机制而建立的机制，它用来通知用户进程系统事件，而不会引起死锁。最后，3.4节讨论了MINIX新的shutdown序列，该序列由一个新的通知结构创建。

注意，本章之后的其它章节也介绍了内核的几处改进。例如，第四章讨论了MINIX时间管理的变化，第五章涉及了内核的修正，即将特殊的内核任务转移到用户态驱动。

## 3.1 对用户态设备驱动程序的支持

当内核态的设备驱动程序转换成独立的用户态程序时，它们将失去大量曾经拥有的特权。尽管如此，结果实，各种函数由设备驱动程序本身管理，而不是将所有功能重新部署到用户态。例如，用户态的驱动程序不再执行I/O设备。另一个问题是内核数据结构，如进程表和环境变量，不再被直接访问。

因此，MINIX中增加了大量新的系统调用，用来支持用户态驱动程序继续像以前那样工作。驱动程序只要请求内核去做，它们自己则不再做这件事。

这一节讨论了增加的重要的系统调用，附录C中对新增的以及已经存在的系统调用提供了一个详细的列表，同时也提供了一个系统调用如何实现的简要介绍，因此下面将不再讨论。

### 3.1.1 为I/O设备增加的系统调用

用户态程序不能直接读写一个设备控制器的寄存器。在一些计算机上，输入输出是由仅能在内核态执行的特殊指令来完成的。在Pentium上，输入输出是通过输入输出寄存器来完成的，理论上，这些寄存器可以被用户态程序访问。然而，这种机制是不安全的。

三个系统被创建，用来使内核代替用户太驱动程序执行I/O操作。SYS\_DEVIO，最简单的系统调用，允许在某一个时刻对一个单独的I/O端口读写。该调用的一个变种，SYS\_VDEVIO通过传递一个带有（端口，值）对地向量，允许对一系列端口读写。最后，SYS\_SDEVIO允许从一个给定端口读写一个缓冲区。

表3.1：SYS\_DEVIO系统调用的处理函数，这个系统调用仅仅对一个I/O端口，而不是一个用户态的设备驱动程序读或写

|  |
| --- |
| 1 PUBLIC int do\_devio(m\_ptr)  2 register message \*m\_ptr; /\* request message \*/  3 {  4 /\* Perform actual device I/O for byte, word, and long values. \*/  5 if (m\_ptr−>DIO\_REQUEST == DIO\_INPUT) { /\* read I/O port \*/  6 switch (m\_ptr−>DIO\_TYPE) { /\* port granularity \*/  7 case DIO\_BYTE: m\_ptr−>DIO\_VALUE = inb(m\_ptr−>DIO\_PORT); break;  8 case DIO\_WORD: m\_ptr−>DIO\_VALUE = inw(m\_ptr−>DIO\_PORT); break;  9 case DIO\_LONG: m\_ptr−>DIO\_VALUE = inl(m\_ptr−>DIO\_PORT); break;  10 default : return(EINVAL);  11 }  12 } else if (m\_ptr−>DIO\_REQUEST == DIO\_OUTPUT) { /\* write I/O port \*/  13 switch (m\_ptr−>DIO\_TYPE) { /\* port granularity \*/  14 case DIO\_BYTE: outb(m\_ptr−>DIO\_PORT, m\_ptr−>DIO\_VALUE); break;  15 case DIO\_WORD: outw(m\_ptr−>DIO\_PORT, m\_ptr−>DIO\_VALUE); break;  16 case DIO\_LONG: outl(m\_ptr−>DIO\_PORT, m ptr−>DIO\_VALUE); break;  17 default : return(EINVAL);  18 }  19 } else { return(EINVAL); } /∗ illegal request ∗/  20 return(OK);  21 } |

当要求多个连续的设备操作时，为了使系统调用最小化，应该使用一个单独的SYS\_VDEVIO或SYS\_SDEVIO系统调用。尽管重复调用SYS\_DEVIO也是可以的，但这需要更多次数的上下文切换，因此应避免这样做。

I/O设备的系统调用接口被设计为类似I/O函数，尽可能地包含在系统库中。例如声明outb(port,value);可以被s=sys\_outb(port,value);代替，其中s是系统调用的返回值。

### 3.1.2 通用的虚拟拷贝

很多内核依赖性是由设备驱动程序拷贝数据引起的。在MINIX 2.0.4中，这主要是通过物理寻址来完成的。对于内核任务还可以容忍，但是不应该如此信任用户态的设备驱动程序，因此不允许从任何内存地址拷贝，也不允许拷贝到任何内存地址。我们只允许使用虚拟寻址，因为虚拟寻址提供了更多的控制。

MINIX 2.0.4已经有了一个虚拟拷贝函数，但是该函数仅仅用在进程间的文本、数据及堆栈段信息的拷贝。然而，MINIX 2.0.4的分析表明phys\_copy()被用来从用户进程拷贝或拷贝到用户进程、做BIOS I/O并执行RAM的磁盘操作。因此，一个新的通用的虚拟拷贝函数virtual\_copy()被创建，用来支持虚拟寻址的三种类型，表3.2中展示了该函数。而相应得系统调用SYS\_VIRCOPY被重新修正。

表3.2：处理SYS\_VIRCOPY系统调用的通用虚拟拷贝函数。虚拟地址可以在LOCAL\_SEG、REMOTE\_SEG或BIOS\_SEG。该函数使用虚拟寻址从src\_addr拷贝字节到dst\_addr

|  |
| --- |
| 1 PUBLIC int virtual\_copy(src\_addr , dst\_addr, bytes)  2 struct vir\_addr \*src\_addr; /\* source address \*/  3 struct vir\_addr \*dst\_addr; /\* destination address \*/  4 vir\_bytes bytes; /\* of bytes to copy \*/  5 {  6 struct vir\_addr \*vir\_addr[2]; /\* virtual addresses \*/  7 phys\_bytes phys\_addr[2]; /\* absolute addresses \*/  8 int seg\_index; /\* memory segment \*/  9 int i ; /\* SRC or DST \*/  10  11 /\* Copy count should be greater than zero. \*/  12 if (bytes <= 0) return(EDOM);  13  14 /\* Convert virtual addresses to physical addresses. \*/  15 vir\_addr[\_SRC\_] = src\_addr; /\* source virtual \*/  16 vir\_addr[\_DST\_] = dst\_addr; /\* destination virtual \*/  17 for ( i= SRC ; i<= DST ; i++) { /\* convert to physical \*/  18  19 /\* Apply different mapping for different segment types. \*/  20 switch(( vir\_addr[i]−>segment & SEGMENT\_TYPE)) {  21 case LOCAL\_SEG: /\* text, stack or data \*/  22 seg\_index = vir\_addr[i]−>segment & SEGMENT\_INDEX;  23 phys\_addr[i] = umap\_local(proc\_addr(vir\_addr[i]−>proc\_nr),  24 seg\_index, vir\_addr[i]−>offset, bytes);  25 break;  26 case REMOTE\_SEG: /\* far memory areas \*/  27 seg\_index = vir\_addr[i]−>segment & SEGMENT\_INDEX;  28 phys\_addr[i] = umap\_remote(proc\_addr(vir\_addr[i]−>proc\_nr),  29 seg\_index, vir\_addr[i]−>offset, bytes);  30 break;  31 case BIOS\_SEG: /\* BIOS memory area \*/  32 phys\_addr[i] = umap\_bios(proc\_addr(vir\_addr[i]−>proc\_nr),  33 vir\_addr[i]−>offset, bytes);  34 break;  35 default : /\* illegal segment \*/  36 return(EINVAL);  37 }  38  39 /\* Check if mapping succeeded. \*/  40 if (phys\_addr[i] <= 0) return(EFAULT);  41 }  42  43 /\* Now copy bytes between physical addresseses. \*/  44 phys\_copy(phys\_addr[SRC], phys\_addr[DST], (phys\_bytes)bytes);  45 return(OK);  46 } |

在系统库的头文件<minix/syslib.h>中定义了几个宏，用来提供一个方便的系统调用接口。程序员可以使用宏，以代替明确提供虚拟地址的片断。例如，为了从BIOS中读取信息或者在进程间拷贝数据，分别定义了sys\_biosin()和sys\_datacopy()宏。

### 3.1.3 中断处理

当一个硬件中断发生时，处理器自动陷入内核并调用中断服务例程。这个调用是在CPU最高优先级下运行的，因此服务例程可以采取所有必要的行为来处理中断。在MINIX 2.0.4中，设备驱动任务实内核的一部份，因此它们的中断处理程序可以直接被中断服务例程所调用。3.3.1节中会解释这个过程的细节。

因为用户态的驱动程序不能直接处理中断，因此需要在内核中增加一个通用的中断处理程序和一个新的系统调用SYS\_IRQCTL，附录C中给出了该调用的精确参数和选项。用户态驱动程序可以使用系统调用SYS\_IRQCTL在初始化阶段安装一个适当的中断处理策略，然后启动中断。无论中断何时发生，它将由通用的处理程序代替驱动程序来处理中断。

为了确定应当支持什么样的中断策略，我们分析了MINIX 2.0.4中的设备驱动任务的中断处理程序采取的行为。大多数的驱动程序，如PRINTER和AT\_WINI驱动，只读取一个状态寄存器。其它驱动程序，包括TTY驱动，要读取一个端口值并回复一个中断应答。最后，还有一些驱动程序，包括FLOPPY驱动，不执行任何I/O操作，但将所有中断处理留给设备驱动。尽管有些驱动不符合上面的情况，它们可以被重写。

通用的中断处理程序如表3.3所示。当中断发生时，它首先从全局表irqtab中查找策略并执行。支持的中断策略有，（1）怎么也不做，（2）读一个端口值，（3）随机回复一个值并回复一个补位掩码，或（4）向某个端口写一个值。在所有情况下，都将发送一个HARD\_INT通知给设备驱动程序做进一步的处理。最后，策略也需会，也需不会重新启用中断。

表3.3：内核中的通用中断处理程序。当一个硬件中断发生时，它将被根据由用户态设备驱动程序使用SYS\_IRQCTL系统调用所设置的中断策略来处理

|  |
| --- |
| 1 PUBLIC int generic\_handler(hook)  2 irq\_hook\_t \*hook;  3 {  4 /\* Get interrupt policy shorthands for convenience. \*/  5 irq\_policy\_t policy = irqtab[hook−>irq].policy; /\* IRQ policy flags \*/  6 int proc\_nr = irqtab[hook−>irq].proc\_nr; /\* process to notify \*/  7 long port = irqtab[hook−>irq].port; /\* register for I/O \*/  8 phys\_bytes addr = irqtab[hook−>irq].addr; /\* address at driver \*/  9 long mask\_val = irqtab[hook−>irq].mask\_val; /\* bit mask or value \*/  10  11 /\* Read a value from the given port . Possibly echo or strobe it back. \*/  12 if ( policy & (IRQ\_READ\_PORT|IRQ\_STROBE|IRQ\_ECHO\_VAL)) {  13 switch( policy & (IRQ\_BYTE|IRQ\_WORD|IRQ\_LONG)) { /\* port granularity \*/  14 case IRQ\_BYTE: { /\* byte values \*/  15 u8\_t byteval = inb(port);  16 if( policy & IRQ\_STROBE) outb(port, byteval | mask\_val);  17 if( policy & IRQ\_ECHO\_VAL) outb(port, byteval);  18 if( policy & IRQ\_READ\_PORT)  19 phys\_copy(vir2phys(&byteval), addr, sizeof(u8\_t));  20 break;  21 } case IRQ\_WORD: { /\* word values \*/  22 ... /\* like above \*/  23 break;  24 } case IRQ LONG: { /\* long values \*/  25 ... /\* like above \*/  26 break;  27 } default : /\* do nothing \*/ ; /\* wrong type flags \*/  28 }  29 }  30  31 /\* Write a value to some port. Cannot both read and write. \*/  32 else if ( policy & (IRQ\_WRITE\_PORT)) {  33 switch( policy & (IRQ\_BYTE|IRQ\_WORD|IRQ\_LONG)) { /\* port granularity \*/  34 case IRQ\_BYTE: outb(port, (u8\_t)mask\_val); break;  35 case IRQ\_WORD: outw(port, (u16\_t)mask\_val); break;  36 case IRQ\_LONG: outl(port, (u32\_t)mask\_val); break;  37 default : /\* do nothing \*/ ; /\* wrong type flags \*/  38 }  39 }  40  41 /\* Send a HARD INT notification to allow further processing. \*/  42 notify(proc\_nr , HARD\_INT);  43  44 /\* Possibly reenable interrupts depending on the policy given. \*/  45 return( policy & IRQ\_REENABLE);  46 } |

### 3.1.4 获得系统信息

有些设备驱动程序需要知道仅仅在内核中可用的信息。例子包括进程表信息、引导监视参数（boot monitor parameters）及空间内存块的列表。一个新的系统调用SYS\_GETINFO被创建，这样用户进程就可以获得这些信息的一个拷贝。通过拷贝整个数据结构，用户进程能够在这些数据上安全地执行操作。



图3.1：SYS\_GETINFO系统调用支持的最重要的数据结构，对于每个Key，<minix/syslib.h>中定义了一个方便的速记函数

为了得到数据结构的一份拷贝，一个进程必须为该数据结构提供一个Key和一个指针，该指针指向拷贝所放置的位置。不需要指定大小，因为所有的数据结构的大小都是固定的。SYS\_GETINFO调用的内核处理函数见表3.4。图3.1展示了可以获取的内核信息。

表3.4：SYS\_GETINFO系统调用的处理函数。该系统调用请求拷贝一个内核数据结构到调用者的地址空间中的一个给定的地址

|  |
| --- |
| 1 PUBLIC int do\_getinfo(m\_ptr)  2 register message \*m\_ptr; /\* request message \*/  3 {  4 phys\_bytes src\_phys, dst\_phys; /\* abs.copy addresses \*/  5 size\_t length ; /\* # bytes to size \*/  6 int proc\_nr ; /\* process to copy to \*/  7  8 /\* First get the process number and verify it . \*/  9 proc\_nr = (m\_ptr−>I\_PROC\_NR == SELF) ? m\_ptr−>m\_source : m\_ptr−>I PROC\_NR;  10 if (!isokprocn(proc\_nr)) return(EINVAL);  11  12 /\* Set source address and length based on request type. \*/  13 switch(m\_ptr−>I\_REQUEST) {  14 case GET\_PROCTAB: { /\* process table \*/  15 src\_phys = vir2phys(proc); /\* convert to physical \*/  16 length = sizeof(struct proc) \* (NR\_PROCS + NR\_TASKS);  17 break;  18 } case GET\_MONPARAMS: { /\* monitor parameters \*/  19 src\_phys = mon\_params; /\* already is physical \*/  20 length = mon\_parmsize;  21 break;  22 } case GET\_KMESSAGES: { /\* kernel messages \*/  23 src\_phys = vir2phys(&kmess); /\* convert to physical \*/  24 length = sizeof(struct kmessages);  25 break;  26 } case ... : /\* and so on ... \*/  27 ...  28 default : /\* illegal request \*/  29 return(EINVAL);  30 }  31  32 /\* Try to make the actual copy for the requested data. \*/  33 if (m\_ptr−>I\_VAL\_LEN > 0 && length > m\_ptr−>I\_VAL\_LEN) return (E2BIG);  34 dst\_phys = numap\_local(proc\_nr, (vir\_bytes)m\_ptr−>I\_VAL\_PTR, length);  35 if ( src\_phys == 0 || dst\_phys == 0) return(EFAULT);  36 phys\_copy(src\_phys, dst\_phys, length);  37 return(OK);  38 } |

SYS\_GETINFO系统定义的方便速记定义在<minix/syslib.h>。例如，进程表GET\_PROCTAB或监视器参数GET\_MONPARAMS的拷贝分别由sys\_getproctab()和sys\_getmonparams()请求。

SYS\_GETINFO系统调用不仅用来消除内核的依赖性，而且允许一些新的应用。例如，4.1节讨论了一个新的消息服务，它使用了该调用用于调试。5.4.3节讨论了内核诊断如何存储到局部缓冲区和如何由SYS\_GETINFO获得。

### 3.1.5 对支持功能的支持

尽管这一章覆盖了支持用户态驱动程序最重要的系统调用，然而我们还是需要几个其它系统调用，下面简要讨论这几个系统调用。关于MINIX完整的系统调用可参见附录C。

SYS\_EXIT系统调用用来彻底地关闭一个用户态系统服务。该调用主要用户新的关闭序列，3.4节中将讨论关闭序列。此外，该调用还用于万一发生了必须局部处理的错误的情况下。

MEMORY驱动程序使用新的系统调用SYS\_KMALLOC在MINIX系统启动时为一个RAM磁盘分配一块内存，这是一个一次性的静态的分配。该调用返回一个REMOTE\_SEG选择器，用于SYS\_VIRCOPY系统调用。5.2节会讨论用户态的MEMORY驱动程序。

TTY设备驱动程序使用SYS\_PHYS2SEG调用，用于视频RAM。该调用为视频存储增加一个段描述符到TTY驱动程序的区布描述符表（local descriptor table，LDT），授权从用户态直接访问。这将在5.4节中解释。

最后，还有几个已有的系统调用，被更新为用户态驱动程序提供更好的支持。例如，完全修改CLOCK的警报功能，提供了一个新型警报，这会在4.3节中讨论

## 3.2 进程间通信

进程间通信（简称为IPC）允许系统服务彼此协作，并且与内核也互相合作。MINIX中的IPC的特点在于一个基于消息传递的客户端服务方式。这是通过从一个进程拷贝一条请求信息到另一个进程并等待其回应来实现的。这一章讨论了MINIX进程间通信的底层机制并做了各种改进。

### 3.2.1 会合消息传递（Rendezvous message passing）

MINIX交换消息的特点是会合消息传递，会合是一个没有中间缓冲器的双向交互。该交互是完全同步的，这意味着，一个进程向另一个进程发送消息，发送者进程将被阻塞并等待另一个进程准备好，当两个进程都准备好时，消息从发送者进程发送到接收者进程，这样两个进程都能重新投入运行。

尽管会合消息传递的方式比一个缓冲消息传递的方案更容易实现，但是它不是那么灵活，有时甚至不够，因为发送者不得不等待接收者接收消息。当请求系统服务时，阻塞一个用户态进程没有问题，然而，当它想发送一个消息而接收者又没有准备好时，阻塞一个内核任务是让人很不满意的。MINIX 2.4采取了特殊措施预防后者，这将在3.3节中讨论。

这样的后果是，进程间传递的消息的精确顺序对于防止死锁非常重要。尽管内核通过在mini\_send()中扫描目标进程的发送队列来检测死锁，但这并不能解决问题的本质，也就是说，不能解决循环的依赖性问题。如果发现一个死锁，返回一个ELOCKED错误值，但是如果能在结构上预防死锁发生就更好了。MINIX做到了这一点，通过使用一个偏袒的消息顺序，在该顺序中，通信是从顶部开始的。

通常，消息是按一个方向发送的，如果进程X发送消息给Y，那么Y就永远不能发送消息给X。例如，用户进程只能发送消息给内存管理和文件系统服务；包括内存管理和文件系统在内的服务职能发送消息给设备驱动程序和内核任务；依次，驱动程序可以发送消息给内核任务，如CLOCK 或 SYS任务，见2.3节中的图2.4。这个规则也有例外，但是很小心地防止了循环依赖性问题。

### 3.2.2 会合进程间通信的实现

会合消息传递是由系统调用SEND、RECEIVE或两个一起来实现的。调用由一个软中断完成，就是说通过一个INT指令陷入内核。这些都被运行时系统（run-time system, RTS）库函数send()、receive()和sendrec()所隐藏，RTS库可以在src/lib/i\*86/rts/中找到。陷阱被捕作并在内核中由汇编例程s\_call()处理，该例程在src/kernel/mpx.s中定义。

汇编代码保存了机器状态并调用C函数sys\_call()做进一步处理，该函数在src/kernel/proc.c中定义。该函数检查系统调用参数并调用mini\_send()、mini\_rec()或两个都调用来做实际的消息传递。这有可能会在一些方向上影响到调度序列。在mini\_rec()中，当没有消息可用时，调用者进入队列并被阻塞。在mini\_send()中，当一个消息不能发送时才会出现这种情况；如果能发送，那么目标进程将会解除阻塞。完成了系统调用之后，汇编例程s\_call()将重新启动下一个要被调度的进程。

**MINIX原有系统调用实现存在的问题**

对MINIX 2.0.4中系统调用实现的分析表明还存在几个缺点和空间有待改进。首先，在函数sys\_call()中发现一个严重的安全隐患。它保证了用户进程只能使用系统调用sendrec()，这样他们停止等待来自内存管理或文件系统服务的回应，然而，所有其它的进程也能使用send()和receive()函数。虽然这对于保护内存管理和文件系统以防止恶意的用户程序不是一件坏事，然而更重要的是防止内核被阻塞。然而，MINIX 2.0.4的设置允许一个服务通过调用send()而不做相应的receive()阻塞一个内核任务，因为服务是受到信赖的。更多用这种策略的服务需要改变。

此外，MINIX 2.0.4中不能限制进程间通信。除了一个严格编码的保证用户进程只能发送消息到内存管理和文件系统服务的限制之外，所有的通信都是允许的。因此系统服务应该能够处理来自任意进程的意外地请求。

sys\_call()的另一个问题是该函数不检查非法系统调用成员。由于当前设置，一个非法调用成员被RECEIVE系统调用中断。如果没有消息可用，这仅仅阻塞调用者，但是我们可以想象更严重的情节。如果一个请求消息被接收，而调用者没有预料到，那么该请求可能被丢弃而不会发送一个回应，有效地阻塞发送请求的进程。

会合消息传递是同步的这个事实使它很难使用，甚至当进程必须确保消息被立即传递时限的不是那么合适。这是一个异步信任问题。3.3节将讨论内核怎样用特别的处理异步事件的实现来解决这个问题。然而，不属于内核的进程不能使用这个诡计，因此只能希望它们的系统调用成功。

### 3.2.3 MINIX系统调用的修正

完全重写了sys\_call()函数来解决上面描述的缺陷。而实现不阻塞的系统调用，限制进程间以一个通用的方法通信，返回合适的错误代码。新的sys\_call()函数如表3.5所示。

表3.5：完全重写了sys\_call()函数来解决几个缺陷并支持非阻塞得系统调用。它现在返回合适的错误代码，表明内核事件被阻塞，限制任何进程间的通信。非阻塞的系统有一个NON\_BLOCKING标志位

|  |
| --- |
| 1 PUBLIC int sys\_call( call\_nr , src\_dst , m\_ptr)  2 int call\_nr ; /\* SEND, RECEIVE, BOTH \*/  3 int src\_dst ; /\* source or destination \*/  4 message \*m\_ptr; /\* message in caller’s space \*/  5 {  6 register struct proc \*caller\_ptr = proc\_ptr ;  7 int req\_function = call\_nr & SYSCALL\_FUNC;  8 int may\_block = !(call\_nr & NON\_BLOCKING);  9 int mask\_entry; /\* bit in send mask \*/  10 int result ; /\* system call result \*/  11  12 /\* Protect the system by requiring caller to await the result . \*/  13 if((iskernel(src\_dst) || isuserp(caller\_ptr)) && req\_function != BOTH) {  14 result = ECALLDENIED; /\* BOTH was required \*/  15 }  16 /\* Verify that source/ destination process is valid . \*/  17 else if (! isoksrc\_dst(src\_dst)) {  18 result = EBADSRCDST; /\* nonexistent process \*/  19 }  20 /\* Check if the request is known and try to perform it . \*/  21 else {  22 switch(req\_function ) {  23 case SEND: /\* send a message \*/  24 /\* Fall through, SEND is done in BOTH. \*/  25 case BOTH: /\* send and receive \*/  26 if (!isalive(src\_dst)) {  27 result = EDEADDST; /\* cannot send if dead \*/  28 break;  29 }  30 mask\_entry = isuser(src\_dst) ? USER\_PROC\_NR:src\_dst;  31 if(!isallowed(caller\_ptr−>p\_sendmask, mask\_entry)) {  32 result = ECALLDENIED; /\* denied by send mask \*/  33 break;  34 }  35 result = mini\_send(caller\_ptr, src\_dst, m\_ptr, may\_block);  36 if(req\_function == SEND || result != OK) {  37 break; /\* done, or SEND failed \*/  38 } /\* BOTH falls through \*/  39 case RECEIVE: /\* receive a message \*/  40 result = mini\_rec(caller\_ptr, src\_dst, m\_ptr, may\_block);  41 break;  42 default :  43 result = EBADCALL; /\* illegal system call \*/  44 }  45 }  46 return( result ); /\* system call status \*/  47 } |

现在一个小但重要的检查确保了内核任务不再被阻塞，对所有的向内核的调用要求使用sendrec()，这样内核任务总是能回应。这类似于检查调用者是否是一个用户进程。不再允许使用send()或receive()，这将导致一个ECALLDENIED错误。继续保持检查src\_dst是否是一个有效的进程号，但是如果检测出问题的话现在返回EBADSRCDST。

下一步是看是否知道系统调用并试图执行请求。这是由一个switch语句来完成的，switch语句使代码更易读并容易增加新的系统调用，这是很有必要的。在MINIX 2.0.4中，仅存在的系统调用便是发送和接收消息。然而，现在支持非阻塞变量NB\_SEND和NB\_RECEIVE，这将在下面讨论。此外，现在非法的系统调用被检测出来并导致EBADCALL错误。

sys\_call()的新特性是发送掩码，该掩码对可能通信的进程提供细致的控制。发送掩码事每个进程的位码，表明一个给定的进程发送。它们在src/kernel/sendmask.h中定义。如果一个SEND或BOTH系统调用完成，检测src\_dst的位是否在调用者的发送码中置位，否则返回错误ECALLDENIED。在下面的保护机制中将讨论更多的细节。

**非阻塞系统调用**

MINIX为了防止缓冲消息的麻烦不提供异步消息传递，因为非阻塞的消息传递相对容易实现。该实现基于一个位NON\_BLOCKING，可以用户系统调用号中。这个标志位同两个拆开系统调用号的位掩码SYSCALL\_FLAGS和SYSCALL\_FUNC定义在<minix/com.h>中。

send()和receive()的非阻塞变体分别叫做nb\_send()和nb\_receive()。这些函数的原型定义在<minix/syslib.h>，RTS库，src/lib/i\*86/rts/\_sendrec.s，被更新来包含了这部分实现。

sys\_call()中，被请求的消息传递函数是通过用SYSCALL\_FUNC对系统调用号做掩码来确定的。指示这个标志位是否被置位的一个布尔值被传递到mini\_send()和mini\_rec()。如果NON\_BLOCKING位被设置并且源进程或目标进程没有准备好，那么mini\_send()和mini\_rec()将返回ENOTREADY，而不是阻塞调用者。

非阻塞系统调用的变体也可以用在阻塞的特殊情况中。这对于用户态设备驱动程序和不能检查进程表中另一个进程状态的服务而言特别有用。现在使用的替代方法是polling。nb\_send()和nb\_receive()调用怎样用户MINIX的新的关闭序列的例子见3.4节和4.1节。

**系统调用保护**

代替仅仅限制用户进程，实现了一个更通用的保护机制。所有进程现在有了一个新的进程表入口，存储一个发送码用以确定它们被允许发送到那些进程。如果一个进程想发送消息给其它进程，首先要检查进程的发送码中相应的位是否被置位。

注意到没有单独的接收码表明一个给定进程可以从那些进程接收消息，只能通过发送码确定。一方面，如果进程A可以发送消息给进程B，那么假设B被允许接收A的消息。另一方面，如果进程B不被允许从A接收消息，那么A的发送码中关于B的位没有被置位，因此A不能发送消息给B。

发送码中对于所有系统进程都有明确的位，就是说，内核任务、设备驱动、服务和INIT都同用户进程一样有一个相应的位。因为所有的系统服务都是静态地包含在系统映像中并且有一个已知的进程号，因此发送码是确定的。

所有系统服务的实际发送码定义保存在一个单独的文件src/kernel/sendmask.h中。发送码是在两个默认的掩码帮助下定义的，这是两个位操作，用来允许或禁止向个别进程发送消息。

在src/kernel/table.c中，新定义的发送码放置于映像表中，该映像表用于MINIX系统启动时在main()中建立初始进程表。这样所有的系统服务就自动有了它们的发送码。

用户进程的发送码是用不同的方法设置的。用户进程总是由一个叫做SYS\_FORK的系统进程创建的，它们的发送码在SYS任务的函数do\_fork()中被设置为USER\_PROC\_SENDMASK。类似，如果一个用户进程退出，将在do\_xit()种清理进程表项并将发送码设置为DENY\_ALL\_MASK。

一个临时的异常被创建来使网络服务INET运行。这个服务始于一个普通的用户进程，但依靠SYS\_SVRCTL系统调用升级为一个服务。因为用户进程的发送码对于INET服务来说太严格了，因此在系统调用的处理函数中将给与它更多的权力。这个异常时临时的，ALLOW\_ALL\_MASK位被设置。

对于系统服务的动态控制，一个更好的接口定义是将来研究的一部份。例如，SYS\_SVRCTL函数必须更新，增加一个动态设置发送码的机制。

**消息传递函数的改变**

函数mini\_send()和mini\_rec()也在某些方面更新了。首先，这两个函数从sys\_call()接收了另一个参数，sys\_call()说明系统调用是否可以阻塞。如果发送源和发送目标都没有准备好，那么是不孕许阻塞的，返回ENOTREADY。

在MINIX 2.0.4，mini\_send()包含一个严格编码的检查来确保用户进程只能发送消息给文件系统和内存管理服务。这已经被上面讨论的sys\_call()函数种通用的检查所取代。

函数mini\_rec()在两方面被修改，它包含一部分之优在内存管理服务阻塞的情况下才执行的代码，用来检测是否存在应该在某一时候发送到内存管理服务的悬而未决的内核信号 。由于增加了一个新的通知内存管理关于悬而未决信号的方法，以上检查就没有必要了，这将在下面讨论。此外，对阻塞中断的检查被推广为检查任意时间的阻塞通知。这两点修改都将在下一章节中详细讨论。

## 3.3异步事件的处理

MINIX的的大部分通信是从上层开始，内核处理诸如定时器和硬件中断的事件，必须是由高一层的进程所通知。因为这些系统事件相对于MINIX的会合消息传递机制来说是异步发生的，因此必须采取特别的方法防止这个问题。在前一节中，我们已经讨论了MINIX使用一个偏袒的消息顺序来预防思索。另外，内核中又使用了一个特殊的结构确保颠倒顺序的消息不能阻塞内核。

需要向上层进程发送消息的异步事件包括硬件中断、异常、内核信号、警报和关闭MINIX。下面讨论每个事件的原始的和新的实现。

### 3.3.1 原始实现

硬件中断发生在I/O设备中时，它们必须被处理。一旦处理器接收到一个中断请求（interrupt request，IRQ），它就要停止当前的指令，调用与中断请求向量相应得中断服务例程。这些例程位于src/kernel/mpx.s，在系统启动时导入中断向量表。服务例程在栈上保存全部状态，并执行中断处理程序，中断处理程序是由处理IRQ线的设备驱动程序注册的。中断处理程序通常不做什么工作，只是通知设备驱动程序去做进一步的处理。由于硬件中断的异步性，当在中断处理程序内传递消息时，可能会发生竞争条件。例如，硬件中断可以嵌套，可能干扰其它进程切换函数，如sys\_call()和sched()。因此，一个特殊的函数interrupt()被用来通过一个HARD\_INT消息通知设备驱动程序发生了中断。如果在interrupt()发现了一个竞争条件，调用放进‘held’队列，等待下一次非竞争条件重启由unhold()激活。如果驱动程序已经准备好接收HARD\_INT消息，将直接传递，并且驱动程序被调度运行。否则，消息将被标记为‘blocked’，然后当驱动程序以HARDWARE或ANY调用receive()时就传递消息。一旦中断处理程序结束，服务例程就通过重启下一个要调度的进程返回。

异常处理类似于硬件中断。当捕获到异常时，处理器切换上下文并执行src/kernel/mpx.中的一个异常处理程序。接着一场处理程序以异常号为参数调用src/kernel/exception.c中的函数exception()。用户进程中的异常转换为信号处理，而系统中的异常将引起一个panic()并关闭MINIX。下面会讨论信号处理和MINIX的关闭。

各种情况下都可能发生信号，但它们总是在内核通过给内存管理服务发送一个消息被处理。通常的会合消息传递用在这，但仅仅是已经确认内存管理服务准备好接收消息的情况下。大部分的工作是由src/kernel/system.c中的两个函数来完成的。如果一个进程被发送信号，函数cause\_sig()更新改进程的信号掩码并设置PENDING标志位。然后它仔细检查内存管理服务是否能被通知得到。如果内存管理服务空闲并在等待消息，函数inform()将由cause\_sig()直接调用。否则，内存管理阻塞等待消息并且信号味觉，则由mini\_rec()处理。函数inform()通过队每个进程发送一个带有标志位PENDING的消息将所有未决信号推向内存管理服务。

当一个警报定时器终止时，警报发生在CLOCK任务中。MINIX 2.0.4有两种类型的警报，叫做用户警报和同步警报。设置和终止警报的函数在src/kernel/clock.c中定义。当一个用户警报终止时，CLOCK引起一个SIGALRM信号。这个处理和所有其它内核信号的处理方式一样。当一个同步警报终止时，CLOCK委托同步警报任务SYN\_AL为代表，该任务发送一个CLOCK\_INT消息给请求警报的进程。CLOCK任务在interrupt()函数的帮助下通知SYN\_AL任务，这个它不能阻塞。不幸的是，SYN\_AL任务不能使用这个函数通知请求警报的进程，改为调用一个普通的模块send()。

最后，关闭或重启从原则上说也是一个异步事件，应该通知给所有进程允许它们干净退出。然而，MINIX 2.0.4没有这样做。关闭代码包含在src/kernel/keyboard.c中的函数wreboot()种。它通过调用stop\_task()函数停止内核中的一系列任务，然后关闭MINIX。用户态的系统服务被简单忽略。

**原始实现存在的问题**

不幸的是，MINIX 2.0.4是用一种不同的方法处理上面描述的每个异步事件。描述的事件是通过特别的实现来解决，而实际上可以用一种通用的方法来解决。此外，还存在一些可能引起严重问题的缺陷。

因为SYN\_AL任务使用普通的send()调用，它将被阻塞，直到请求警报的进程调用了receive()。这种情况很容易延迟其它进程的同步警报请求，或者甚至更糟的话，如果相应的receive()根本没有调用，那么这将永远阻塞SYN\_AL任务。这是一个异步信任的问题，随着依赖于同步警报的进程数量的增多将变得更严重。MINIX只有INET服务使用同步警报，但是当很多设备驱动程序也依赖于同步警报功能时就不合适了，4.2节将会详细解释后者。

MINIX 2.0.4的关闭代码不通知用户态设备驱动和服务，因为直接的函数调用不可能穿越地址空间，并且MINIX 2.0.4没有一个消息传递机构能安全地执行这项任务。这是由问题的，因为重要的系统进程没有被告知MINIX关闭，这样就不能运行它们的清除代码。例如，当MINIX由于一个panic()而带来关闭的时候，文件系统不能同步，因而可能会丢失数据。

下一节描述了处理异步事件的一个统一的方法，能解决所有的这些问题。

### 3.3.2 一个新的通知结构

设计了一个新的消息传递结构notify()用于以一种统一的方式将异步系统事件通知给所有类型的进程。最初，notify()函数是放在interrupt()函数之后的，但是后来它们并入一个单一的函数，可以用于所有的系统通知。最后函数被命名为notify()，因此这个名字能更好表达该函数的作用，见表3.6。

表3.6：notify()函数，允许内核任务安全地将任一种类型的系统事件通知给所有类型的进程，而不必担心被阻塞。通过将调用放置于held对列可以处理竞争条件。如果一个进程没有准备好接收通知，记录这个阻塞

|  |
| --- |
| 1 PUBLIC void notify(proc\_nr, notify\_type )  2 int proc\_nr ; /\* process to notify \*/  3 int notify\_type ; /\* type of notification \*/  4 {  5 register struct proc \*rp; /\* pointer to process \*/  6 message m; /\* notification message \*/  7 unsigned int notify\_bit ; /\* bit in bit mask \*/  8  9 /\* See if notification type is known. \*/  10 notify\_bit = (unsigned int)(notify\_type − NOTIFICATION);  11 if (notify\_bit >= NR\_NOTIFICATIONS) { /\* check validity \*/  12 panic("Incorrect notification type", notify\_bit);  13 return;  14 }  15 rp = proc\_addr(proc\_nr);  16  17 /\* Check for races with other process switching functions . \*/  18 if (k\_reenter != 0 || switching) {  19 lock (); /\* disable interrupts \*/  20 if (!rp−>p\_notify\_held) { /\* add to held queue? \*/  21 if (held\_head != NIL\_PROC) held\_tail−>p\_nextheld = rp;  22 else held\_head = rp;  23 held\_tail = rp;  24 rp−>p\_nextheld = NIL\_PROC;  25 }  26 set\_bit(rp−>p\_notify\_held, notify\_bit); /\* update held mask \*/  27 unlock (); /\* reenable interrupts \*/  28 return ; /\* retry later \*/  29 }  30 switching = TRUE; /\* protect rest of call \*/  31  32 /\* If process is not ready to receive HARDWARE message, record the blockage. \*/  33 if(!isreceiving(rp−>p\_flags) || !isrxhardware(rp−>p\_getfrom)) {  34 set\_bit(rp−>p\_ntf\_blocked, notify\_bit); /\* update blocked mask \*/  35 switching = FALSE; /\* end protection \*/  36 return ; /\* handle in mini rec \*/  37 }  38  39 /\* Destination is awaiting message. Send notification and announce process ready. \*/  40 m.m\_source = HARDWARE; /\* construct message \*/  41 m.m\_type = notify\_type; /\* set notification \*/  42 CopyMess(HARDWARE, proc\_addr(HARDWARE),&m,rp,rp−>p\_messbuf);  43 rp−>p\_flags &= "RECEIVING"; /\* message delivered \*/  44 clear\_bit(rp−>p\_ntf\_blocked, notify\_bit); /\* update blocked mask \*/  45 ready(rp); /\* receiver now can run \*/  46 pick\_proc(); /\* schedule new process \*/  47 switching = FALSE; /\* end protection \*/  48 } |

3.3.1节中会讨论，新的函数notify()与interrupt()有类似的语义。它向内核任务提供了一个机制，实现了非阻塞的消息传递而不用缓冲消息。然而，与MINIX 2.0.4中的函数interrupt()相比，新函数notify()提供了重要的扩展功能。首先，它可以用来通知任意类型的进程而不仅仅是内核任务。其次，除了硬件中断外，它支持所有类型的系统事件的通知，包括上面所讨论的。

为了防止混乱，我们要注意，notify()结构并不是一个如send()或receive()一样的系统调用，相反，notify()是围绕send()来建立的并确保它不会因为检查接收进程的状态而阻塞。因此，它只能用在内核。

**具体实现细节**

该实现要求对MINIX源代码作一些小的改变。src/kernel/proc.h中的进程结构被更新，包括为‘held’和‘blocked’通知的位掩码。在<minix/com.h>中增加定义所有通知类型，它们与NOTIFICATION相关，防止与已经存在的消息类型冲突。src/kernel/proc.c中的函数notify()、unhold()和mini\_recv()中有所实现。与MINIX 2.0.4最大的不同在于同志的不同类型被区分，CopyMess()用来向任意类型的进程拷贝通知消息。

notify()函数解决潜在的竞争条件，这样它能够安全地用于中断处理程序和任务级。如果一个通知与其它进程切换函数竞争，如系统调用和硬件中断，则将目标进程的‘held’位掩码的一个为置位并将调用放入‘held’对列。在下一个非竞争时刻unhold()将会激活held通知。它被通用地支持各种不同类型的通知。

此外，notify()函数是非阻塞的。如果接收者没有准备好接收一个通知消息，则将目标进程‘blocked’位掩码的一个为置位。当目标进程调用receive()时，会检查这个位掩码。如果发现了一个被阻塞得通知，则局部构造一个通知消息并复制给接收者。

尽管‘held’和‘blocked’位码消除了缓冲消息传递的必要，但还是有一个小的交换。每个通知类型的只有一个位，不可能存储额外的信息，如通知消息的发送人或参数等。因此，可以传递的仅有信息就是通知类型，并且所有的通知消息都以HARDWARE作为它们的消息源。4.3节将讨论，这个特性对于在用户态设备驱动程序的异常处理非常有用。

### 3.3.3 异步事件的处理

处理硬件中断基本上和以前一样。HARD\_INT消息类型被重定义为一个有效的通知类型。中断处理程序现在调用notify(proc\_nr, HARD\_INT)来向设备驱动程序报警关于硬件中断，而不是去调用interrupt(proc\_nr)。这样做有一个很积极的作用，那就是代码变得更易读了。

通过除去几个异常情况，MINIX的信号处理被大大地简化了。定义了一个新的通知类型KSIG\_PENDING来代替KSIG消息类型。检查内存管理服务是否准备好接受消息的代码可以从cause\_sig()和mini\_recv()可以去掉了，这个工作现在由cause\_sig()中一个单独的notify(MM\_PROC\_NR, KSIG\_PENDING)调用来完成。inform()函数，用来一旦准备好接收信号时把未决的信号推给内存管理服务，也可以去掉。在新的方法下，当接收了一个KSIG\_PENDING通知时，内存管理服务为一个未决的信号反复推动内核。于是这就把处理信号的责任交给内存管理服务。

通过去除SYN\_AL任务解决了同步警报任务的问题，当警报中止时，CLOCK任务不再调用SYN\_AL任务对send()进行一个潜在地阻塞调用。相反，因为新的函数notify()不会阻塞，因此现在CLOCK任务敢向通过发送SYN\_ALARM通知来请求警报的进程报警。

最后，MINIX关闭序列被完全修正，为此定了一个HARD\_STOP类型。进程表现在被活跃的系统服务扫描，而不是直接的函数调用。每个活跃的进程被通知即将关闭，并被允许某一时刻清理。根据不同类型的进程间存在的依赖性将通知排序，这样底层的进程可以继续服务于顶层的进程。3.4节中描述了新的关闭序列。

## 3.4 一个新的关闭序列

MINIX的关闭序列在前面的章节中已作为一个请求不符合通信顺序的异步事件的例子简要讨论了。这里，我们将进一步研究细节，并解释干净关闭MINIX系统的新的关闭序列和所有系统服务。

### 3.4.1 原始实现

MINIX 2.0.4的关闭序列是直接的，它被几个事件触发。例如，用户可以通过按‘Ctrl-Alt-Del’组合键或输入‘shutdown’或‘reboot’来关闭系统。一个内核惊恐也能引起系统关闭。如果触发关闭的事件在内核外被检测到，SYS\_ABORT系统调用被用来通知内核。例如，内存管理服务使用该调用来处理关闭。

在各种情况，内核在src/kernel/keyboard.c中的wreboot()运行关闭代码。该函数首先将所有的IRQ向量标志，这样设备驱动程序就不会再被中断。然后它允许几个设备驱动程序通过调用stop\_task()函数运行它们的清理代码。例如，floppy\_stop()停止软盘驱动器运转，cons\_stop()将主控制台选择为可视。

If the reason to shut down was a panic, the user is allowed to make debug

dumps before rebooting MINIX. The debug dumps are easily handled in the

wreboot() function, because all needed functions and data structures are in

the kernel address space. The functions that make the actual debug dumps

are contained in src/kernel/dmp.c.

Finally, wreboot() brings down MINIX. The action that is taken depends

on the system’s environment and the reason to shutdown that contained

in the argument that is passed to wreboot(). Known values are contained

in <unistd.h>. RBT\_MONITOR, RBT\_PANIC, RBT\_REBOOT, and RBT\_HALT all

try to return to the boot monitor with level0(monitor) and possibly run some

code at the boot monitor. If it is not possible to return to the boot monitor

because the environment variable mon return is 0 or if how is RBT\_RESET, a

hard reset is issued with level0(reset).

**原始设计存在的问题**

The most important issue with the original setup has already been discussed

in Section 3.3. The shutdown code in MINIX 2.0.4 does not notify userspace

device drivers and servers. This is problematic because important

system services are not informed about MINIX’ shutdown and cannot run

their cleanup code. The FS, for example, cannot synchronize when MINIX

is brought down due to a kernel panic and thus is likely to loose data.

Another problem is that the shutdown code in MINIX 2.0.4 is responsible

for quite some dependencies when device drivers are moved out of the kernel.

The stop\_task() functions can no longer be called for user-space drivers, because

function calls across address spaces are not possible without bypassing

the MMU’s normal protection mechanisms.

Finally, the location of the shutdown code is not logical at all. Although

shutting down is an important, system-wide event, the shutdown code in

MINIX 2.0.4 is part of the TTY driver. The historic reason for this may be

that the the ‘Ctrl-Alt-Del’ command is issued via the keyboard, but this

does not reflect the other ways to shutdown MINIX. In most cases, the MM

initiates a shutdown via a sys\_abort() system call.

### 3.4.2 关闭MINIX的新设置

MINIX’ shutdown was completely redesigned to solve all problems that are

listed above. The new shutdown sequence heavily relies on the new notify()

message passing construct that was discussed in Section 3.3. This solves the

problem relating to the dependencies and allows to inform all system services

about MINIX’ shutdown, instead of kernel tasks only. The new shutdown

sequence is illustrated in Figure 3.2.



图3.2：

Figure 3.2: MINIX’ new shutdown sequences notifies all user-space system services

in order of possible dependencies. The TTY is notified last so that it can output

diagnostic messages during the shutdown sequence.

The shutdown sequence based on three functions that are executed in

turn: prepare\_shutdown(), stop sequence() and finally shutdown(). These functions

are discussed below. Because shutting down is the opposite of starting

MINIX the new code was placed in src/kernel/main.c.

**准备MINXI的关闭**

The function prepare\_shutdown() that initiates MINIX’ shutdown sequence is

shown in Listing 3.7. The function first checks a global flag, shutting\_down,

to prevent recursive shutdowns. This could, for example, happen when the

MM or FS panics during their cleanup. It then disables most interrupts,

like wreboot() does in MINIX 2.0.4. The CLOCK\_IRQ is excluded, however,

because the hardware clock is needed by the CLOCK task to keep track of

the watchdog timers that are used for the stop sequence.

Listing 3.7: The new function prepare\_shutdown() disables most interrupts, checks

if a panic occurred and whether debugging dumps should be allowed, and starts

the stop sequence unless a CPU exception was raised.

表3.7：

|  |
| --- |
| 1 PUBLIC void prepare\_shutdown(how)  2 int how; /\* reason to shut down \*/  3 {  4 if(shutting down) return; /\* shutdown yet active \*/  5  6 /\* Mask interrupts, but keep clock ticking for timers . \*/  7 outb(INT\_CTLMASK, (1<<CLOCK\_IRQ));  8  9 /\* See if a panic occured and debug dumps are possible. \*/  10 if(how == RBT\_PANIC) { /\* a panic occurred \*/  11 message m; /\* assemble a message \*/  12 m.m\_type = PANIC\_DUMPS;  13 if(nb\_send(TTY, &m) == OK) /\* try , but don’t block \*/  14 return ; /\* await abort from TTY \*/  15 }  16  17 /\* Now start MINIX’ shutdown. Try to stop system services. \*/  18 notify(TTY, HARD\_STOP); /\* to primary console \*/  19 shutting\_down = TRUE; /\* set shutdown active \*/  20 tmr\_arg(&shutdown\_timer)−>ta\_int = how; /\* pass shutdown reason \*/  21  22 if(skip\_stop\_sequence) { /\* set in exception () \*/  23 kprintf("Fatal exception; skipping stop sequence.\n", NO\_ARG);  24 shutdown(&shutdown\_timer); /\* directly shut down \*/  25 } else {  26 kprintf("Notifying services about MINIX’ shutdown.\n", NO\_ARG);  27 stop\_sequence(&shutdown\_timer); /\* run stop sequence \*/  28 }  29 } |

The next step is to check whether MINIX is brought down due to a panic.

If this is the case and if the TTY driver is ready the user is allowed to make

debug dumps before shutting down. Debug dumps are discussed in Section

4.1. The kernel carefully verifies that the TTY driver is still alive and

ready by using the new, nonblocking nb\_ send() function. The function prepare

shutdown() returns after sending a PANIC\_DUMPS request message so that

the TTY can run. Control is only given up if the message is successfully sent.

The TTY is responsible to restart the shutdown sequence with a SYS\_ABORT

system call when the user is done. More information on debugging dumps

on a panic can be found in Subsection 4.1.4.

Finally, prepare\_shutdown() sets the flag shutting\_down to inform other kernel

part about the upcoming shutdown. Then it decides if stop\_sequence()

should be run. Naturally, the stop sequence should be run whenever possible.

However, it is skipped to prevent doing more harm once a CPU

exception was raised for a kernel process. Segmentation faults, for example,

tend to persist when the stop sequence gets to run because kernel tasks are

nonpreemptable and are scheduled with the highest priority.

**停止所有系统服务**

The function stop\_sequence() that tries to cleanly stop all system services

before shutting down is shown in Listing 3.8. All system services are notified

with a HARD\_STOP message and given some time to clean up and exit. The

notifications are sent in the order of possible dependencies, starting at the

highest level and ending at the lowest. This means that the FS server, for

example, can still use the AT\_WINI driver to cleanly shutdown. This is

illustrated in Figure 3.2. Unfortunately, dependencies between similar types

of processes are not resolved this way. Therefore, the TTY driver was placed

at a lower level than other other device drivers.

Listing 3.8: The new function stop\_sequence() tries to cleanly stop all system

services before shutting down. Only when all processes have been exited—either

gracefully or forcibly—MINIX is really shut down.

表3.8：

|  |
| --- |
| 1 PUBLIC void stop\_sequence(tp)  2 timer\_t \*tp ;  3 {  4 static int level = PPRLSERVER; /\* highest level first \*/  5 static struct proc \*p = NIL\_PROC; /\* next process to stop \*/  6 static char \*types [] = /\* all process types \*/  7 {"task", "system", "driver", "server", "user"};  8  9 /\* See if previous process exited . Else force it to exit . \*/  10 if (p!= NIL\_PROC) { /\* skip first time \*/  11 kprintf("[%s]\n", isalivep(p) ? "FAILED" : "OK");  12 if (isalivep(p)) /\* check if exited \*/  13 clear\_proc(p−>p\_nr); /\* force process exit \*/  14 }  15  16 /\* Lookup next process to exit . Shutdown when all done. \*/  17 if(p == NIL\_PROC)p = BEG\_PROC\_ADDR;  18 while(TRUE) { /\* stop all processes \*/  19 if(isalivep(p) && p−>p\_type == level) { /\* next process found \*/  20 kprintf("− Stopping %s ", p−>p\_name);  21 kprintf("%s ... ", types[p−>p\_type]);  22 shutdown\_process = p; /\* used in sys exit () \*/  23 notify(proc\_number(p), HARD\_STOP); /\* alert the process \*/  24 set\_timer(tp,get\_uptime() + STOP\_TICKS, stop\_sequence);  25 return ; /\* allow it to stop \*/  26 }  27 p++; /\* check next process \*/  28 if(p >= END\_PROC\_ADDR) { /\* this level done? \*/  29 level = level − 1; /\* go to next level \*/  30 p = BEG\_PROC\_ADDR; /\* restart at begin \*/  31 if (level == PPRI\_TASK) { /\* tasks remain alive \*/  32 set\_timer(tp, get\_uptime() + HZ, shutdown);  33 return ; /\* display output \*/  34 }  35 }  36 }  37 } |

When stop\_sequence() is run for the first time, it starts at the beginning of

the process table to find the next process that must be stopped. If a process

is found its name and type are printed for the user’s interest. A global variable,

shutdown\_process, is set to indicate which process is being stopped. This

makes it possible to immediately continue the stop sequence once the process

has exited. A watchdog timer with stop\_sequence() as a watchdog function is

set to make sure that the stop sequence continues within STOP\_TICKS ticks

if the process does not exit voluntarily. Then stop\_sequence() returns so that

the process to be stopped can be scheduled and can run its cleanup code.

Now two things can happen. If the process that is being stopped notices

the HARD\_STOP message it can clean up and stop with a SYS\_EXIT system

call. The function do\_exit() that handles this system call continues the stop

sequence if the shutting\_down flag is set and the process that exited indeed is

the shutdown\_process. If the process does not exit within the timeout interval,

the watchdog timer expires and will run its watchdog function.

In both cases, the stop\_sequence() function is run again. If the process to

be stopped exited gracefully this is reported to the user. If a disobedience is

detected, the failure is reported and the process is forcibly exited. Then the

stop sequence looks up the next process to stop. It continues where the last

search ended and proceeds to next level if there are no more active process

at the current level.

Finally, if the last level has been processed, that is, if all processes have

exited, MINIX is brought down with a call to shutdown(). This is done by

setting a watchdog timer to give the user some time to inspect the status

of the entire shutdown sequence. The watchdog timer’s argument is used to

pass the shutdown status as before.

It has to be noted that the TTY driver must be the last process in this

sequence so that it can display the stop status of each process at the primary

console. The TTY thus resides at the end of the lowest level of user processes.

When the TTY driver finally receive a HARD\_STOP notification it directly

displays its own stop status before exiting.

**关闭MINIX**

The last function, shutdown(), deals with returning to the boot monitor or

doing a hard system reset. It contains the majority of the code that was

part of wreboot() in MINIX 2.0.4. This code was not changed compared to

the discussion in Subsection 3.4.1. Therefore, it is not treated here.

### 3.4.3 将来的修改

The new shutdown sequence is part of the kernel because system services

have a special status in MINIX and cannot be managed in the same way as

ordinary user processes. System services, for example, cannot do an ordinary

exit() call, which is handled by the MM. Instead, they use the SYS\_EXIT system

call to directly tell the kernel that they want to exit.

As discussed in Section 7.3, there is no obvious reason for this special

treatment. Therefore, the transformation of system services into ordinary

user processes is part of future work. Once this has been done, a large

part of the shutdown code can be removed from the kernel, for example, by

setting up a new server that controls the shutdown sequence. The design

that is presented in this section need not to be changed, though.

# 第四章 新的应用

## 4.1 新的消息服务机制

### 4.1.1 MINIX2.0.4的调试

### 4.1.2 一个分布式的方法

### 4.1.3 观察功能码

### 4.1.4 惊恐之后的调试

## 4.2 看门狗计时器的一般管理

### 4.2.1 原始实现

### 4.2.2 定时器的一般管理

## 4.3 处理无反应硬件

### 4.3.1 MINIX2.0.4的异常处理

### 4.3.2 察觉超时的新方法

# 第五章 消减内核

## 5.1 用户态的PRINTER驱动

### 5.1.1 原始PRINTER任务的修改

### 5.1.2 用户态的PRINTER驱动的安装

### 5.1.3 一般的多级调度

## 5.2 用户态的MEMORY驱动

### 5.2.1 用户态MEMORY驱动的安装

### 5.2.2 读实时时钟

### 5.2.3 ‘ps’效用的问题

## 5.3 用户态的AT\_WINI和FLOPPY驱动

### 5.3.1 检测控制器超时

### 5.3.2 驱动独立代码的改变

### 5.3.3 动态驱动控制器映射

## 5.4 用户态的TTY驱动

### 5.4.1 MINIX关闭代码的重新设计

### 5.4.2 调试

### 5.4.3 输出诊断

# 第六章 相关工作

## 6.1 CMU（Carnegie-Mellon University） Mach

### 6.1.1 内核特性

### 6.1.2 Mach的应用

## 6.2 QNX Neutrino RTOS

### 6.2.1 系统结构

## 6.3 L4微内核API

### 6.3.1 L4的实现

### 6.3.2 L4Ka::Pistachio

### 6.3.3 L4应用的例子

## 6.4 与MINIX的比较

# 第七章 概要和结论

## 7.1 本文的贡献

## 7.2 回顾

## 7.3 将来的工作

# 附录A 源代码树的组织

## A1 内核文件总览

## A2 新源代码树的组织

# 附录B 怎样应用改变（How to apply changes）

## B1 增加程序到系统映像中

## B2 增加系统调用

## B3 增加系统库

# 附录C MINIX的系统调用

## C1 系统调用实现的组织

## C2 MINIX3.0.0中系统调用总览