华中科技大学

硕士学位论文

嵌入式实时操作系统——ARTs-OS的I/O管理

姓名：严昌浩

申请学位级别：硕士

专业：计算机软件与理论

指导教师：张文彬

2002.5.13

摘要

在国家自然科学基金的资助下，研究目标为：研制一个全新的，基于微内核的实时嵌入式操作系统（ARTs-OS），将仅涉及输入/输出（I/O）部分。

一般认为，I/O部分约占操作系统70%的代码量。并且，I/O体系结构是操作系统中最为繁杂的部分，但对此却一直缺乏足够而详尽的研究。

在吸收了部分软件体系结构的思想，并在设计、实现ARTs-OS中I/O部分的具体工作中发现：连接种类的多样性（特别是有硬件中断时）是I/O结构复杂的原因，并依据这一规律，通过改变某些连接的类型，成功的降低了ARTs-OS中I/O结构的复杂度。同时，含混笼统的I/O体系结构实际上可以由四个要素精确的定义，它们是：中断处理方式、存在方式、存在位置、同步和异步。因此，I/O设计转化为对四大要素的取值。它表明，设计一个全新的I/O体系结构也并非完全的随意和自由，它能且仅能为四个要素的某种组合，而且其中一些是不可能的。应用该设计思想完成了ARTs-OS的I/O详细设计，并可以对ARTs-OS的I/O系统的效率、实时性、复杂度等性能指标进行事前的评估。

另外一个研究重点是实时磁盘调度算法的性能评价。传统上，衡量实时磁盘调度算法的优劣，除了（模拟）试验外，再没有什么更好的办法。文中认为微观调度决定宏观性能，由此，提出了一种新的调度评价模型。应用该模型，从理论上可以证明：在理性调度下，实时性能和效率必定是矛盾的。并用试验数据证实：EDF算法不是一个实时性能好的算法。

最终实现的ARTs-OS是一个拥有动态加载、核外驱动、软/硬中断可选等许多高级I/O特性的系统。在最后，对实现上的一些具体技术难点作了分析，并对这些难点的各种解决方案作了比较，并给出了部分ARTs-OS的实现细节。

关键词：操作系统；实时；嵌入式；输入/输出；软件体系结构；磁盘调度算法

ABSTRACT

Under the subsidization of National Nature Science Fund, the research work concentrates on a real-time embedded operating system (ARTs-OS) with micro-kernel infrastructure. Only the Input/Output system (I/O) is involved in this paper.

Generally, the code lines of I/O occupy nearly 70% of those of an OS. And the architecture of I/O is the most chaotic one than that of others. But it is always lacking deep and enough research.

Based on the opinion of software architecture and the I/O architecture of ARTs-OS, the complexity of connection types (especially with hardware interrupt) is the basic cause of the complexity of I/O. according this principle, the complexity of ARTs-OS's I/O has been decreased successfully by changing some connections' types. And that the obscure definition of I/O architecture can be precisely confined in four factors which are: how hardware interrupt be handled, how device drivers exist, where drivers locate and how drivers handle synchronous and asynchronous relations. So, I/O design can be converted to make choices in these four factors. And at the same time, it shows that even a new I/O architecture can not be designed freely. It can and only can be one combination of these factors some of which are solutions impossible. The I/O architecture of ARTs-OS is designed under the principles, and some indexes such as efficiency, real-time satisfaction and complexity of ARTs-OS can be evaluated beforehand.

Another important emphasis is evaluation of real-time disk schedule algorithms. Traditionally, there is no other better way to evaluate schedule algorithms but experiments. A new evaluation model is proposed based on that macro performance is decided by micro schedule. Under this model, a conclusion that if a schedule algorithm is good enough, we can not possess the real-time performance and efficiency at the same time can be derived. And the experimental data under this model show that EDF is not a good algorithm.

The implementation of I/O architecture of ARTs-OS includes several advance features, such as: dynamic loader, out kernel driver and soft-interrupt/hard-interrupt option, etc. But in order to implement these features, some special technology difficulties emerge. Finally, a few solutions and their cons and pros attributes are discussed, and the implementation of ARTs-OS is described on source code level.

Keywords: operating system; real time; embedded system; I/O; software architecture; disk schedule algorithms

1 绪论

1.1 操作系统中 I/O 的研究

一般认为，操作系统由五大部分组成：进程/线程调度机制、内存管理、进程间通讯（Inter-Process Communication IPC）、中断管理和输入/输出（I/O）。其中，I/O 在操作系统中占有重要的地位[1]。但是在体系结构上，I/O 部分远没有其它部分的结构清晰和统一，几乎每一个操作系统都有它们独特的 I/O 构架。I/O 成为操作系统中最为凌乱的部分[2]。遗憾的是，很少有论文对 I/O 的体系结构进行专门而细致地研究。

目前，国内外对 I/O 的研究一般有如下几个方面：

1. 磁盘调度策略研究

设计磁盘调度算法，使其满足性能要求。如：非实时系统中应满足效率、吞吐量、等待时间的要求；实时系统应首先满足实时性要求等。大量论文对此做了详尽的分析[3-6]。

2. 缓冲区管理策略

数据从物理设备到应用程序空间一般要经过 I/O 驱动程序的缓冲区[7]。缓冲是用来平滑 I/O 请求峰值的技术。缓冲区管理策略是研究缓冲区的分配和释放算法。分配涉及何时分配以及分配的大小，释放涉及何时释放缓冲区以及释放谁的算法。

另外有一种无缓冲技术也是当前研究的热点[8,9]。在无缓冲的情况下，数据从物理设备直接到达应用程序空间，从而消除了从系统缓冲区复制到用户程序空间的这一过程。这一技术称为减少“I/O 步长”，MACH 操作系统对此有特别的研究[10,11]。

3. 动态加载技术

I/O 设备的加载可分为三个级别：

(1) 源代码级加载。需要重新编译操作系统生成映象。如：早期的 Linux 操作系统。

(2) 静态可加载。需要修改配置文件，重新启动操作系统。如：Windows 操作系统。

(3) 动态可加载。在操作系统运行的情况下，可加载/卸载I/O驱动程序。如：Linux2.2及更高版本；Windows 2000 Server也有一定的动态加载功能。很明显，动态加载技术对于支持可热插拔设备的嵌入式应用来讲是一个关键特性。它的实现技术成为I/O结构研究的热点之一[12,13]。

4. 核外I/O技术

在通用操作系统中，I/O作为内核的一部分而存在。但核外I/O（User level I/O）有其特别的优势：它能够避免昂贵的上下文切换；保护模式的出入；数据流在不同级间的拷贝。而且，它具有：灵活、有效、可移植等特点。但有效地实现核外I/O有其特定的困难，许多论文阐述了基于各自的操作系统的具体实现方法[14-17]。

5. 微内核技术

微内核（Micro-kernel Architecture）作为现代操作系统的典型特征之一，它区别于传统的巨大内核（monolithic kernel）是其内核仅仅包括基本的内存管理、IPC和最基本的调度机制，而大量的系统服务以进程/线程方式运行，并且，IPC是它们之间通讯的唯一信道[18]。

1.2 实时操作系统中的实时I/O技术

1.2.1 实时操作系统的特殊要求

文献[19,20]指出，实时操作系统应具备以下5方面的要求：

1. 确定性：操作系统的确定性是指它可以按照固定的、预先确定的时间或者时间间隔执行操作。它取决于响应中断的速度和实时处理能力。

2. 响应性：是指在知道中断后，操作系统为中断提供服务的时间。它包括：最初处理中断并开始执行中断服务例程（ISR）所需要的时间总量；执行ISR所需要的时间总量；中断嵌套的影响。

3. 用户控制：在实时系统中，允许用户细粒度地控制任务优先级是必不可少的。用户应能够区分硬实时和软实时，并对每一类中确定相对优先级。

4. 可靠性：实时系统是实时地响应和控制事件，它的暂时故障往往不能够通过系统重启来解决。因为，灾难可能已经发生，重启系统已经没有任何意义。

5. 故障弱化运行：故障弱化操作（fail－soft operation）指系统在故障时尽可能多地保存其权能和数据的能力。故障弱化运行的一个重要特征是稳定性。当系统不能完成所有任务时，它总是优先完成最重要的任务。

1.2.2 实时操作系统的典型特征

当前的实时操作系统典型地包括以下特征[21,22]：

1. 快速的进程/线程切换；
2. 体积小（当仅有最小功能集时）；
3. 迅速响应外部中断的能力；
4. 通过信号量、信号和事件等进程间通讯工具，实现多任务处理；
5. 使用特殊的顺序文件，可以快速的存储数据；
6. 基于优先级的剥夺式调度；
7. 最小化禁止中断的时间间隔；
8. 用于使任务延迟一段固定的时间或暂停/恢复任务的原语；
9. 特别的报警和超时设定。

实时系统的中心问题是短程任务调度程序。在设计这种调度程序时，公平性和最小平均响应时间并不是最重要的，最重要的是所有硬实时任务都在它们的最后期限内完成（或开始），尽可能多的软实时任务也可以在它们的最后期限内完成（或开始）。

大多数当代操作系统都不能直接处理最后期限，它们设计成尽可能地对实时任务作出响应，使得当临近最后期限时，一个任务能够迅速地被调度。从这一点看，实时应用程序在许多条件下都要求确定性的响应时间在几毫秒道小于 1 毫秒的范围内。前沿应用程序，如军用飞机模拟器，通常要求响应时间在 10 到 100 微秒的范围内。

1.2.3 实时 I/O 的特征

1. 实时磁盘技术

非实时的磁盘调度算法一般以吞吐量和平均等待时间为衡量标准，而在实时 I/O 中，这一标准并不适用。更重要的是 I/O 请求的截止期、实时优先级等要求。实时磁盘技术的核心就是研究实时磁盘的调度算法。

2. 实时文件系统

在文件系统的结构上，考虑实时响应性的相关技术。

3. 防止优先级逆转技术

防止优先级逆转是实时 I/O 技术的一个特定的要求[23]。优先级逆转是指，由于低优先级的进程/线程占有 I/O 设备，使高优先级的得不到设备而等待，并且，由于该低优先级的进程/线程在其它资源的争夺中经常处于被阻塞的地位，导致高优先级的进程/线程通过 I/O 设备的传递而阻塞在一个可能比它低的进程/线程上。优先级逆转在实时系统中是必须加以防止的[24]。

1.3 实时 I/O 分析中的软件体系结构思想

软件体系结构是对大尺度软件的一种描述和分析方法[25,26]。它主要研究：由部件（component）组成系统的组织结构；全局控制结构；通讯、同步、数据访问的协议；设计元素的功能分配；设计元素的组成；物理分布；伸缩性和性能；进化方向；在设计方案中选择等等。一个特定的系统定义为部件以及部件间的联系——连接件的集合[27]。

当前，国内外对体系结构研究的主要方面有[28,29]：

1. 体系结构形式化描述。如：Wright[30]，ACME[31] 等，给出了连接件的标记式形式化描述方法，但它们的直观性差，Wright 标记法有一定的局限性[32]。

2. 体系结构风格的归纳整理[33,34]。提炼出若干种模式（pattern）和风格（style），形成类似工程手册一样的经验，供后来的设计者使用。这是一种实用化的研究路线。

3. 特定领域软件体系结构研究[35]。其通用性差。

4. 其它。如：文献[32]提出，采用主动连接件这一种连接形式。虽然它有助于连接件的实体化。但它将连接件与主动机制联系，必然导致本已十分复杂的连接件更加复杂。

本文对 I/O 体系结构的分析思想就是来自软件工程中的软件体系结构分析方法。

1.4 本文研究的主要内容

本文分上下两篇。上篇——结构篇包括第二、三章。下篇——实现篇包括第四、五章。上篇主要对 ARTs-OS 中 I/O 的软件结

构进行总体分析和详细设计，其中采用的分析方法也可以推广到其它操作系统的 I/O 设计中。下篇主要是阐述实现实时 I/O 的一些理论和技术。

在结构篇中，第二章主要是从整体上分析 ARTs-OS 的 I/O 逻辑结构，阐述 I/O 结构复杂性的原因，并提出了降低 ARTs-OS 的 I/O 结构复杂性的可能途径。第三章通过对 ARTs-OS 的 I/O 构架的详细设计，提出了 I/O 体系结构的确切内涵——四要素结构。并由此对 I/O 结构的所有可能组合作了详尽的分析。

在实现篇中，第四章提出了一种新的实时磁盘算法的评价模型。它具有很好的评价能力，并对设计新算法也有帮助。第五章涉及 ARTs-OS 的动态加载、核外驱动等高级 I/O 特性的实现技术细节。

2 ARTs-OS 中 I/O 的总体结构设计

ARTs-OS 的 I/O 总体结构的设计目标定为：

1. I/O 结构应当满足操作系统的实时性要求。

2. 为了避免传统操作系统在 I/O 结构上的凌乱和复杂性，特别对 ARTs-OS 的 I/O 结构提出应当简洁、清晰的要求。这一点在设计 I/O 的总体结构中作为一个重要的设计准则。

3. 驱动程序的编写和调试简单。因为，驱动程序是驱动程序编写人员和 I/O 系统的界面，设计一个简单和易于调试的驱动程序界面，对于嵌入式操作系统是特别重要的。在嵌入式系统中，I/O 设备复杂、多样，需要用户编写驱动程序的场合远远要比通用操作系统中要多，所以对驱动程序界面的简洁性的要求，要比通用操作系统更高。

在本章中，将介绍 ARTs-OS 的 I/O 结构的总体设计。通过仔细分析了传统操作系统(LINXU 和 WINDOWS NT)产生 I/O 结构复杂的原因，有针对性地提出了降低 I/O 结构的复杂性的途径，并利用该方法于 ARTs-OS 结构设计中，使 ARTs-OS 中 I/O 结构的复杂性降低到相当低的程度。

2.1 ARTs-OS 的 I/O 逻辑结构

不同操作系统之间的 I/O 结构差异很大。在设计一个全新的操作系统 I/O 结构之前，分析现有成熟的操作系统的 I/O 结构将是非常有益的事情。

2.1.1 Linux 与 NT 的 I/O 逻辑结构图

1. Linux 的 I/O 逻辑结构图

从图 2－1 中，我们可以看出 Linux 中 I/O 结构的特点：

(1) 缓冲区高速缓存，它不依赖于任何文件系统，Linux 缓冲区立于底层介质和设备驱动程序而存在，Linux 对高速缓冲区的分配/释放是采用 LRU 算法；

(2) 所有块设备的读写操作请求，通过标准的内核程序提交给设备驱动程序，其中有（唯一设备标识符，块编号），但设备驱动程序读写的目标均是高速缓冲区区域；

(3) 缓冲区将数据刷新到物理块设备上，是由 BDFLUSH 和 UPDATE 的核心线程在必要时完成的，或者在系统关机时完成；

(4) 缓冲区与用户进程之间，可以用内存对内存的 DMA 传输；

(5) 支持动态的模块装/卸。

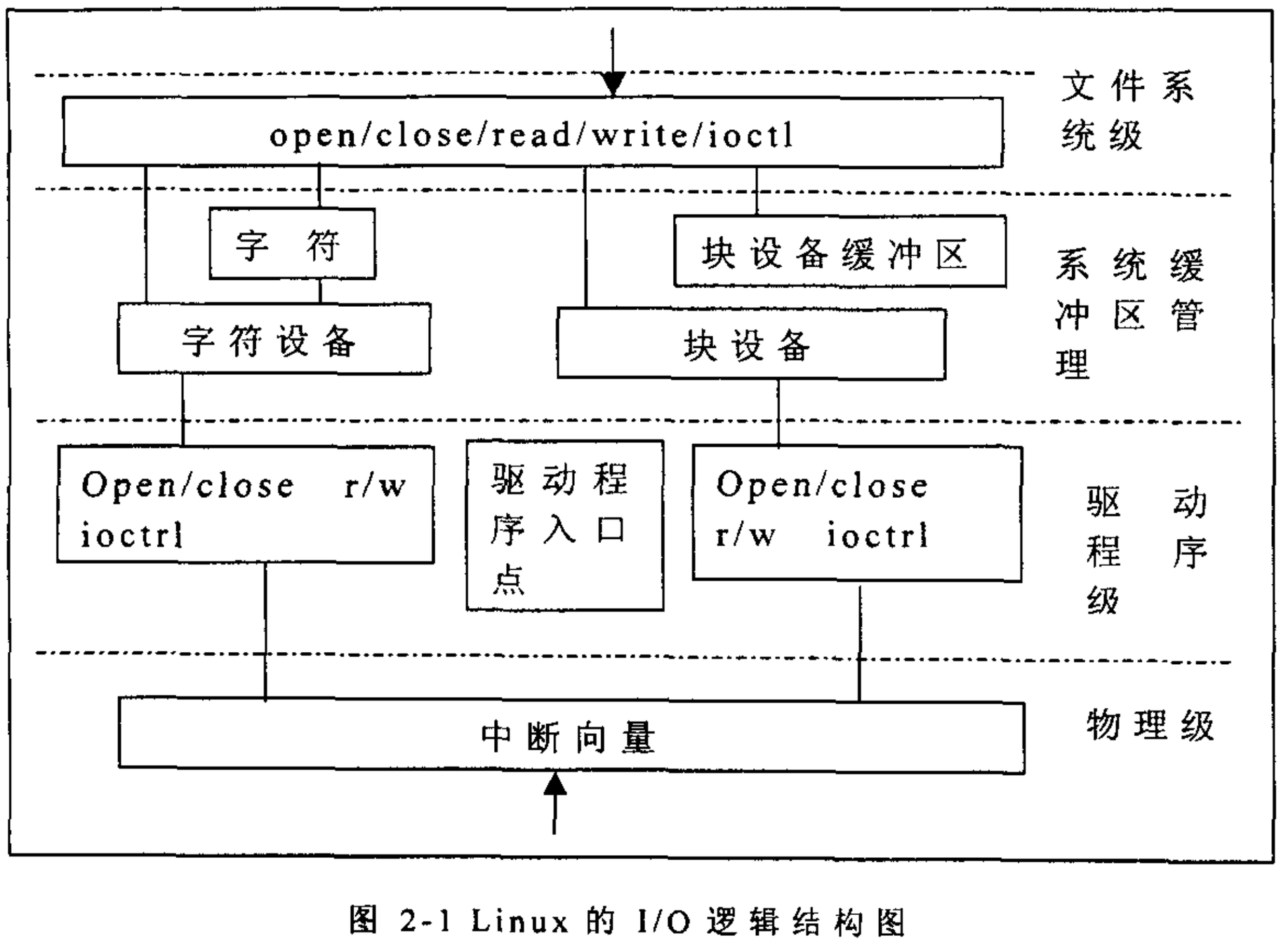

图 2-1 Linux 的 I/O 逻辑结构图

从图 2－1 中，我们还可以明显看到，在 Linux 的 I/O 结构中将设备区分为字符和块设备（实际上，还有网络设备也是单独的一类）。并且，它们的处理方式不同，主要集中在缓冲区的管理上。字符设备用的是简单的环形字符缓冲区，而在块设备中，使用的是块缓冲池，系统内核对块设备的缓冲区有特别的支持。虽然，这在一定程度上简化了驱动程序的设计，但同时，它也加大了驱动程序与内核之间的联系强度，使得驱动程序与内核的关系更加紧密。若要实现核外驱动、动态加载等高级特性时，这将成为不利的因素。

2. WINDOWS NT 的 I/O 逻辑结构图

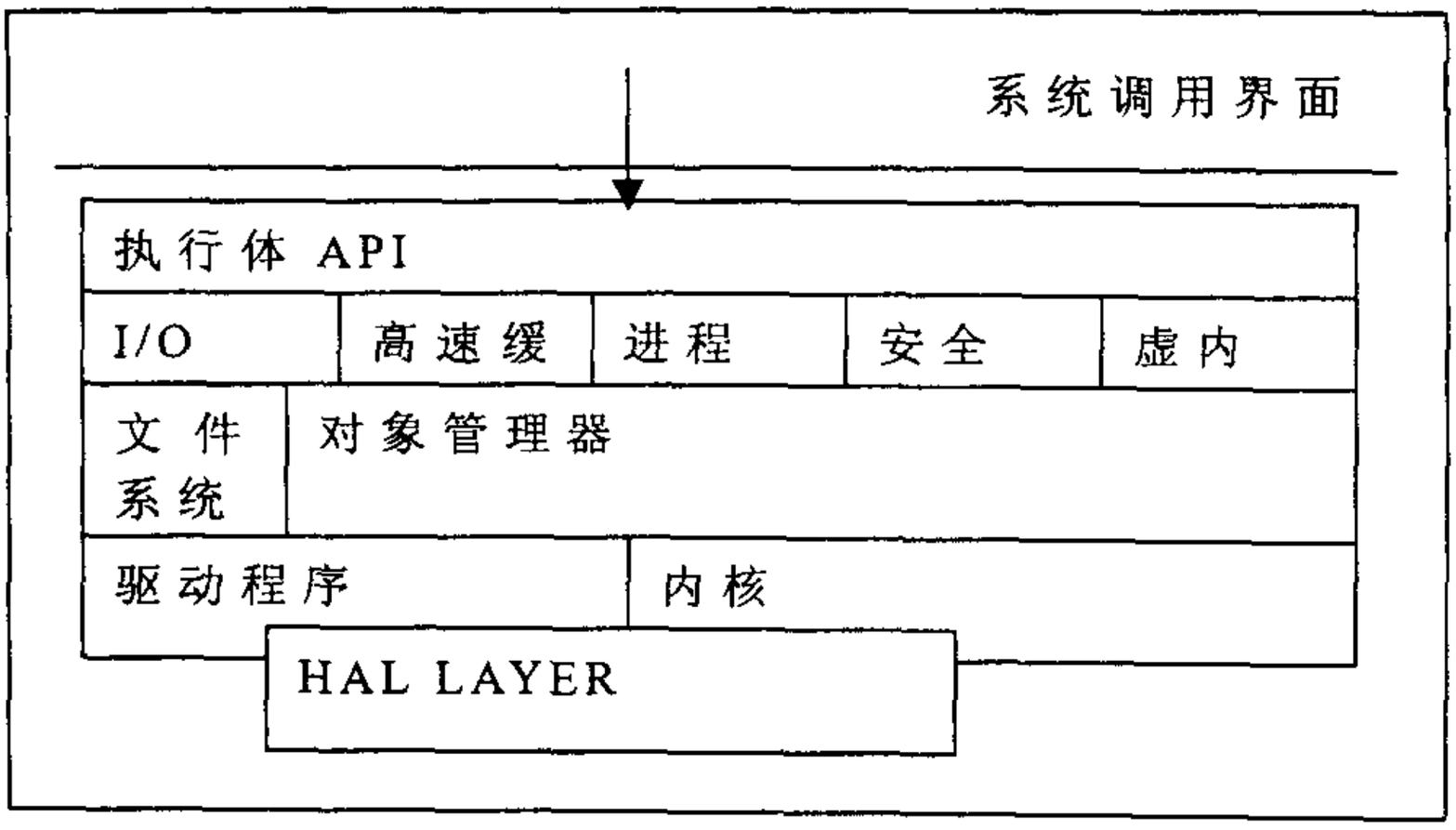

图 2－2 WINDOWS NT 的 I/O 逻辑结构图

WINDOWS NT 的 I/O 结构的典型特征为：

1. 由 IRP（I/O REQUEST PACKET）包驱动，而 DPC 用于弹出设备驱动程序 IRP 堆栈中的 IRP；

2. 支持同步/异步 I/O，由于本质上是同步，效率低，而异步 I/O 实际上是由 I/O 管理器进行排序而顺序执行；

3. 单独而全面的高速缓冲机制；

4. 支持所有的文件系统，消除重复代码；

5. 由用户和 OS 共同控制文件的某一个部分是否在缓冲区中；

6. 可恢复的文件系统。

2.1.2 ARTs-OS 的逻辑结构

一般地，ARTs-OS 的 I/O 体系结构可以抽象为图 2－3。

从图 2－3 中，我们可以看出，ARTs-OS 的 I/O 体系结构主要涉及四个部件：

1. I/O 实体：驱动程序，是完成 I/O 任务的组织者；

2. I/O 硬件：完成物理的 I/O 操作；

3. 内核：驱动程序需要内核的支持（内存、中断管理等），同时也为内核提供服务，如：虚存的调页；

4. 应用程序：大部分 I/O 请求的发起者。

从它们的连接关系上分为两大类：数据类连接和控制/调用类

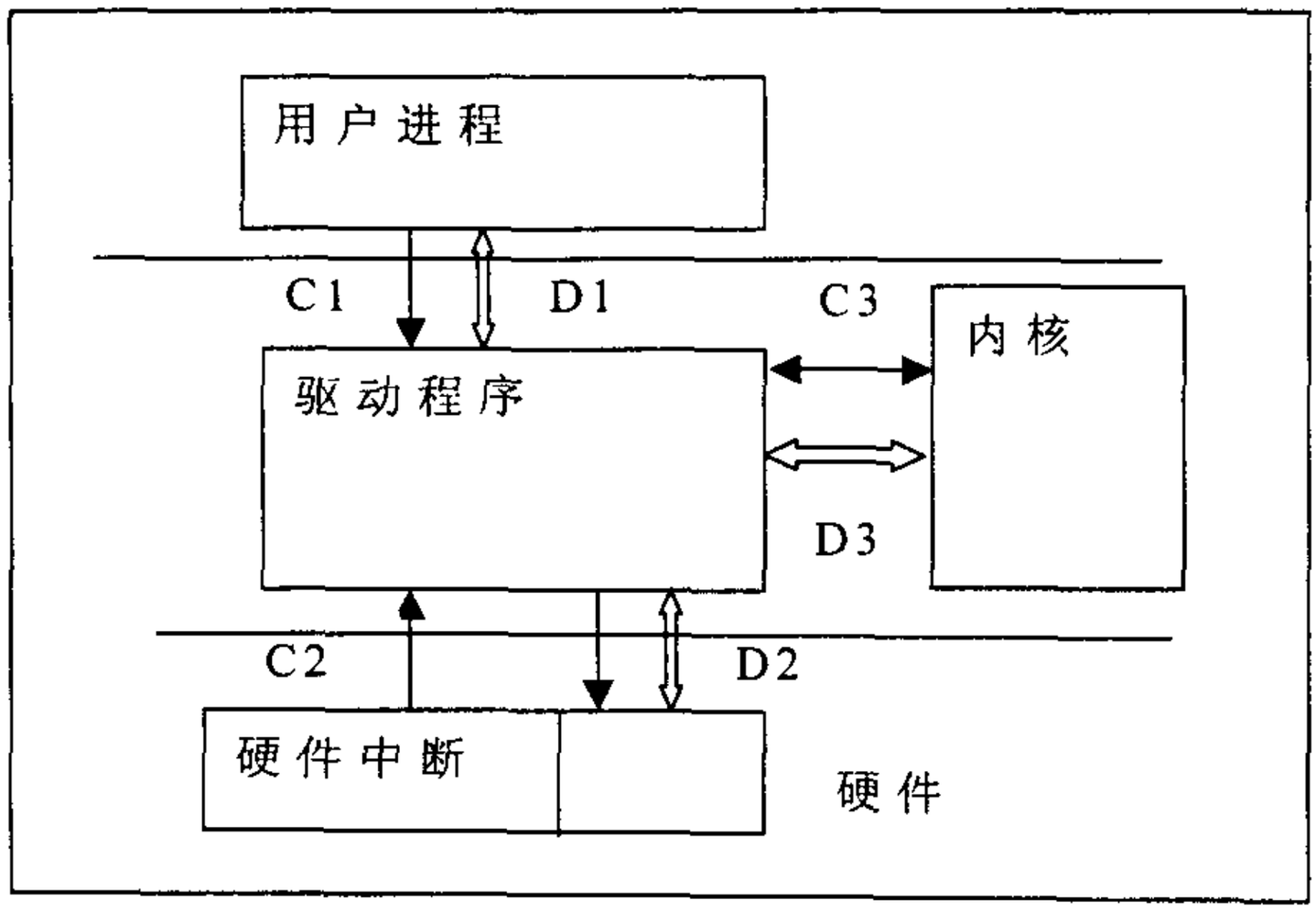

图 2-3 ARTs-OS 的 I/O 体系结构图

连接。

1. 数据类连接

应用程序和驱动程序之间的双向数据流动（用 D1 表示，下类似）、硬件和驱动程序之间的双向数据流动（D2）、核心与驱动程序之间的双向数据流动（D3）。不同的箭头方向代表相应的读/写操作。

2. 控制/调用类连接

从应用程序到驱动程序的调用关系（C1）、从硬件中断到驱动程序的调用关系(C2)、驱动程序和内核的双向调用关系(C3)。表面上 C1、C2、C3 它们均属于同一类型的连接，但更深入的分析表明：C1 与 C3 是同一类型，但 C2 却是另一类型的连接。但在此图中却无法表达这一点，这需要对连接作更仔细地分析。

其实，图 2－3 中的 I/O 体系结构是存在进一步降低其结构复杂性的途径的。如下的几节将分析产生结构复杂性的原因，并找到合理地降低其结构复杂性的方法。

2.2 降低 ARTs-OS 的 I/O 结构复杂性

2.2.1 操作系统中 I/O 的结构复杂性的存在

操作系统的 I/O 体系结构在实体上反映为对驱动程序的设计。而一般地，驱动程序给人印象是：驱动程序一般很短小，完成基本 I/O 功能的驱动程序都可以在 1000 行 C 语句内(外加少量汇编语言）完成。但是，驱动程序涉及的内容很繁杂，它往往要与 OS 的内核、硬件、用户 API 界面、中断等等有着千丝万缕的联系，使得若按照代码行的单价计算，驱动程序的代码成为往往成为应用系统中最为“昂贵”的代码。具体地讲，造成这一现象的原因可能有如下几点：

1. 设计驱动程序需要对操作系统内核有较深入的了解。通用操作系统（如：Linux、Windows 等）的驱动程序都处在系统内核之中，它们与内核有很多的控制信息和数据信息的交换，不同操作系统有不同的交换方式，使得程序员学习量很大。

2. 驱动程序属于底层硬件编程，需要应用大量的特权函数或较复杂的系统调用。例如，驱动程序一般均需要 I/O 指令、内存映像、核内/核外空间转化、中断处理例程等特殊指令或调用，增加了驱动程序自身的复杂性。

3. 驱动程序调试困难。很少有调试工具能对驱动程序实现源代码级的调试。

4. 驱动程序的稳定、可靠直接关系到系统的稳定性。因而，这对程序员写出可靠、“无错”代码提出了更高的要求。

但是，更深入的分析表明，以上这些都是驱动程序复杂的表面现象，而真正原因不是驱动程序本身有多么的复杂，而是由驱动程序背后的操作系统 I/O 体系结构上的凌乱造成的。如果有办法降低其 I/O 结构上的复杂性，就可以极大的简化驱动程序的设计。事实上，降低操作系统的 I/O 结构复杂性的方法是存在的。本论文的原型操作系统 ARTs-OS 的 I/O 构架就是在这种思想指导下设计并实现的。实践表明，驱动程序可以如同普通应用程序一样简单的设计和调试。

2.2.2 评价 I/O 结构复杂性

在前节中说明 I/O 结构复杂性的存在，仅仅是从一种经验的、直观的角度。然而，为了正确地设计一个结构清晰、简单的 I/O 结构，必须弄清楚如下几个最基本的问题：

1. 什么是 I/O 结构的复杂性和 I/O 结构复杂的原因；

2. 如何评价不同 I/O 结构的复杂性。也就是，I/O 结构复杂性的评价原则；

3. 如何降低 I/O 结构复杂性。即：怎样应用合适的的改变结构的方法，降低整体 I/O 结构的复杂性，同时，不能对实时性能、功能等有太大的削弱。

本小节将解决 I/O 结构复杂性产生的原因和评价 I/O 结构的原则等问题。下小节将进一步阐述降低 ARTs-OS 的 I/O 结构的途径。

1. 连接类型的分类

从实现的角度上，控制连接是指：一个组件对另一个组件存在着激活/调用的联系。这种控制连接的关系可以按照不同的标准分为不同的类型[36]。

从时间角度，控制连接发起者（caller）和被激活者（callee）之间相互协作的关系可分为同步连接和异步连接。同步连接是指：在连接发生时，即：caller 发起连接后，必须等待被 callee 处理请求；而后者也必须在执行完请求后，返回到前者的等待处。其中：caller 的等待方式也及 callee 的返回方式必须是事前约定的。异步连接是指：caller 发出连接请求，callee 被激活，caller 不用等待，同时，callee 也不用返回。例如：我们常见的过程调用是同步连接，中断例程的激活是异步连接。

从空间角度，caller 和 callee 相互协作的关系可分为硬连接和软连接。区分它们的标准是：caller 和 callee 是否在同一个逻辑地址空间。若在同一个地址空间，则为硬连接，否则，为软连接。例如：一般常见的过程调用是硬连接；在 windows 操作系统中，应用程序进程中的消息响应函数，被操作系统的外部消息激活，这样的连接方式是软连接。

从时间和空间的这两种不同角度的分类是正交的。因此，可以产生如下四种连接方式：

（1） 同步硬连接关系：如我们最常用的过程调用模式，在

caller 与 callee 之间，一方面，它们是同步的，另一方面，它们是“直接跳转”。

(2) 同步软连接关系：RPC（远程过程调用），它在语法和语义上与普通的过程调用很相似，但是在 caller 与 callee 之间，存在着命令的缓存机制。

(3) 异步硬连接关系：硬件中断处理系统就是异步硬连接的典型例子，硬件中断发生器与中断响应函数（ISR）之间的控制连接就是这样一种连接关系。

(4) 异步软连接关系：对于某些不需要同步应答的连接，如：一般的 Windows 下的 CallBack 类型的函数就是一个例子。

以上四种连接关系的结构特性有如下几点：

(1) 在硬连接中，caller 和 callee 必须在同一个逻辑地址空间中，也即：空间约束。它表明：处于不同逻辑地址空间中的部件之间，连接的方式只能是软连接，而不可能用硬连接。

(2) 在硬连接中，caller 和 callee 必须在同一个时刻同时存在，也即：时间约束。它表明：时间上不能同时存在的部件之间，连接的方式只能是软连接，而不可能用硬连接。以上两点统称为硬连接的“时空约束”。

(3) 软连接存在不确定的连接时延。连接时延是指：caller 发出连接请求的时刻与 callee 启动执行请求的时刻之间的差的绝对值。硬连接的时延是可以事先确定的。并且，一般硬连接的时延要小于软连接的时延。

(4) 软连接在实现上，存在命令缓冲机制。因此，callee 必须存在消息泵机制。消息泵是指：在部件中存在的一种类似死循环的检索消息队列的循环结构。异步软连接中，calee 有消息泵，而同步软连接中，双方均应有消息泵。

(5) 异步连接的双方，必须在不同的执行调度单位（如：线程 / 进程）中。否则，如果在同一个调度单位内，则它们的执行序列将是确定的，而不是异步的。

(6) 异步 callee 在执行时刻，不能对其所处的上下文环境作任何假设。

(7) 软连接必定存在（消息名，实义行为）的抽象映射。这降低了 caller 与 callee 之间的耦合度。

2. I/O 体系结构复杂性的评价原则

基于上述连接件分类方法，对一个 I/O 系统的体系结构复杂性，提出如下评价原则：

原则一 一个软件系统的结构复杂性，并不取决于系统中连接件的总数量，而是，系统中连接件种类的数量。

原则二 同步连接比异步连接结构复杂性要低。

应用这些原则，可以定性的分析软件系统的结构复杂性。

3. I/O 体系结构复杂性评价原则的应用

由原则二，同步连接和异步连接在结构复杂性上的差异是显著的，典型的例子是 Windows 和 Unix 在程序设计上的重大差别。

在 UNIX 下，编程风格是同步的，而在 WINDOWS 下的，它是以消息响应为主，围绕消息响应函数展开，是异步的。这一连接类型上的差异，对程序员界面的影响也是巨大的。我们往往只承认这种差异，但却很少考虑，是什么造成了这种差异。

下面，我们将用连接分类方法和复杂性的评价原则对其进行分析。其实，若把应用程序和操作系统看作两个大的部件，则它们之间的连接图分别为图 2-4 和图 2-5 所示。其中：实直线表示同步硬连接，实折线表示异步硬连接，虚折线表示异步软连接。

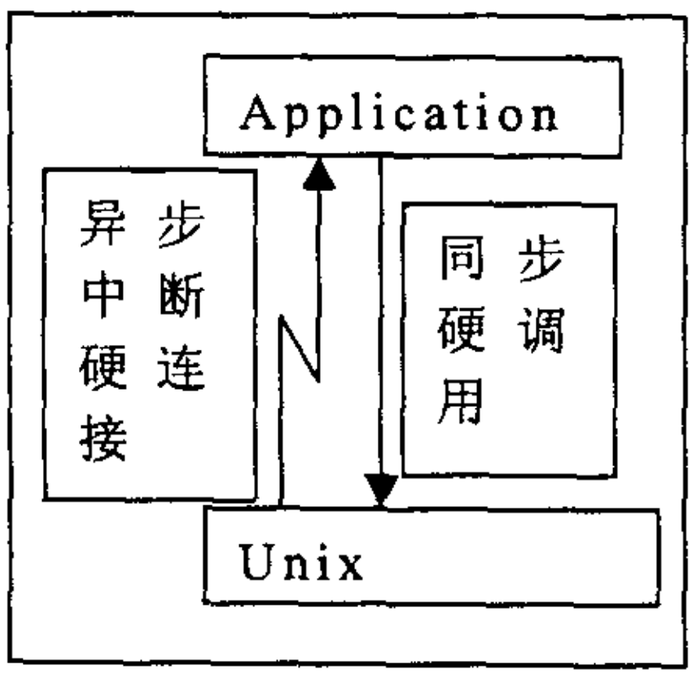

图 2-4 Unix 连接关系

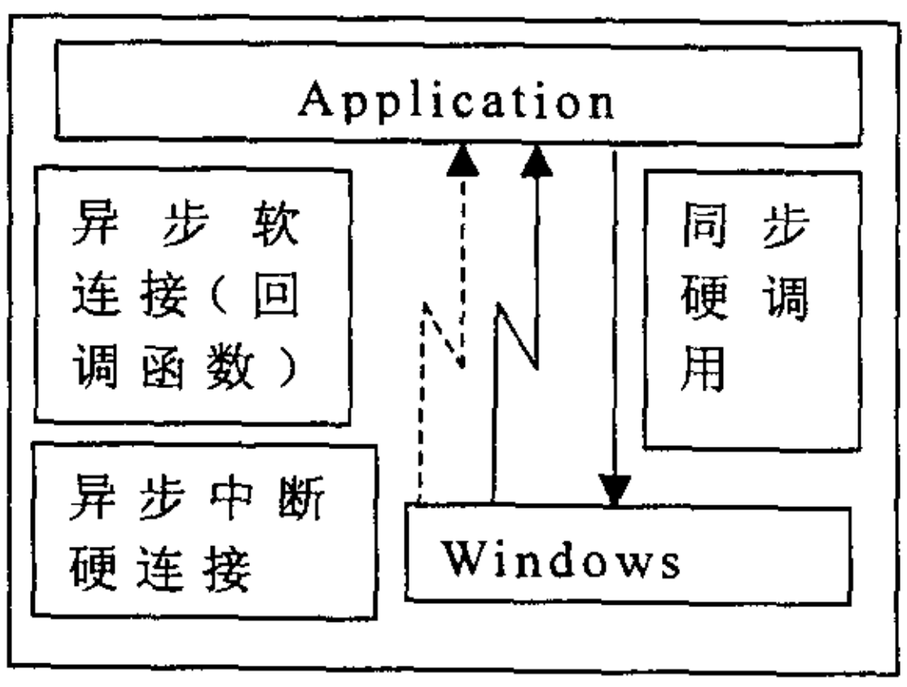

图 2-5 Windows 连接关系

从图 2-4 和图 2-5 中我们可以看出，在 WINDOWS 下，应用程序和操作系统之间的连接比在 UNIX 下多出了一种异步软连接，并且，它在实际编程中占的比重很大。因此，仅从结构上讲，WINDOWS 编程要比 UNIX 复杂。同理，我们也可以看出，驱动程序由于往往要考虑异步硬连接（硬件中断），因此，它要比普通应用程序在连接件种类上又要多出一个，同样，驱动程序要比普通应用程序复杂。这一点可以从我们的直觉中得到印证。

直观上，同步连接和异步连接的区别是显著的，但软连接和硬连接的区别并不是那么明显，但为什么也还要区分它们呢？同

步硬连接和同步软连接的典型例子分别是：直接过程调用和 RPC。它们虽然在语义上的区别不大，但是，在实现上，RPC 一般是由底层的 RPC 软件包实现，它与直接由计算机硬件实现的过程调用还是有些区别的。例如：在系统失效上，RPC 更容易出现问题，即使没有硬件故障，也可能出现软件通路失效；在系统分布上，由于不需要共享逻辑地址空间，RPC 显然比直接过程调用适应性强；在响应延迟上，硬连接显然要小而且具有确定性等等。这些差别，往往都是一个系统的非功能性需求的重要组成部分。而我们在针对不同的系统需求，构造总体软件体系结构的时候，系统的功能性需求，一般取决于部件的功能，而非功能性需求，就体现在连接件上。也就是说，不同的连接类型，将产生不同的系统非功能性需求。

在计算机系统中，采用软件或是硬件实现，在逻辑功能上，是没有什么差别的，但是，就是这个软硬件划分界面形成了不同的计算机系统结构。同样，在软件体系结构中，部件之间的连接采用软连接或是硬连接，在系统的功能上也没有太大差别，但却构成了不同的软件体系结构。

2.2.3 ARTs-OS 的 I/O 结构复杂性的降低途径

由上节，我们可以知道，操作系统的 I/O 结构复杂性就是 I/O 结构内连接种类的多少。这表明：要降低 I/O 结构复杂性，也就是设法减少 I/O 结构中连接种类的数目（由原则一知），并设法将异步连接改用同步连接（由原则二知）。ARTs-OS 就是在这一想法的指引下，实现了结构复杂性的降低。

通用操作系统（UNIX/WINDOWS）和微内核操作系统的驱动程序与系统的连接关系分别如图 2-6 和图 2-7 所示，其中，在微内核操作系统中，模块间的连接均是采用消息通讯的方式，也就是本文中的软连接的类型。

比较两者，我们发现，采用微内核结构原本是为了增强模块之间的独立性，但在驱动程序中却并没有因此减少它的结构复杂性（它们均存在两种连接方式）。为了简化 I/O 系统结构，一种可行的方法是：将硬件和驱动程序之间的硬异步连接，改为软异步连接，如图 2-8。

这样，驱动程序的结构复杂性就大大降低了，呈现给驱动程序设计员的是这样一个简单的界面：处理各种不同的 I/O 请求（读

/写/打开/关闭/中断响应），其中，中断响应也被简化为一个“普通”的需要处理的请求而已。当然，中断请求的优先级远高于一般的 I/O 请求，而这一点，由系统内核中的带优先级的 IPC 机制对用户（驱动程序员）屏蔽。当然，由于对硬件中断采用软连接处理，会带来中断响应延迟过大的问题，但至少，在系统的结构图上，可以清晰的看到这一点。

ARTs-OS 的 I/O 系统就是采用这样的一种体系结构。但必须注意的是，由前节中各种连接的特性我们可以知道，软连接的连接时延是不确定的。特别是在实时系统中对中断的处理，这一点是非常不利的。为了解决中断响应延迟的问题，我们对 ARTs－OS 的中断服务例程作了局部的调整，仍然可以采用硬异步连接方式，但同时，为了不增加结构的复杂性，对驱动程序员屏蔽了这一细节，使其看到的是统一、清晰的界面。

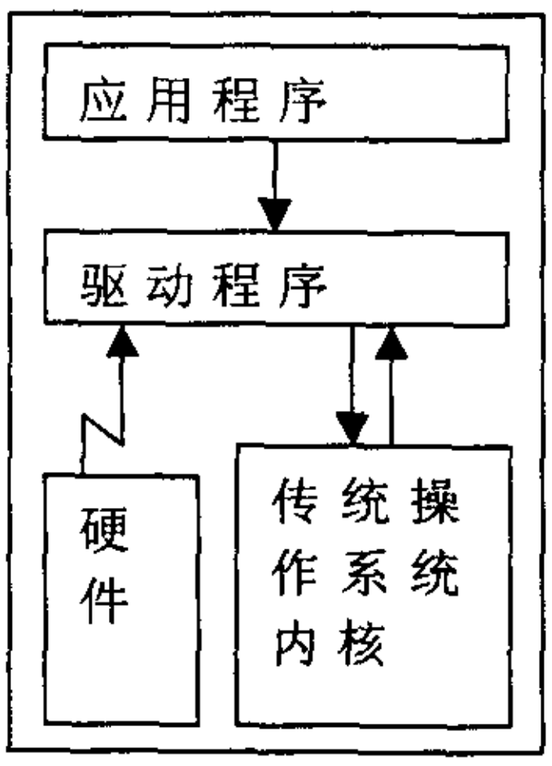

图 2-6 普通 I/O

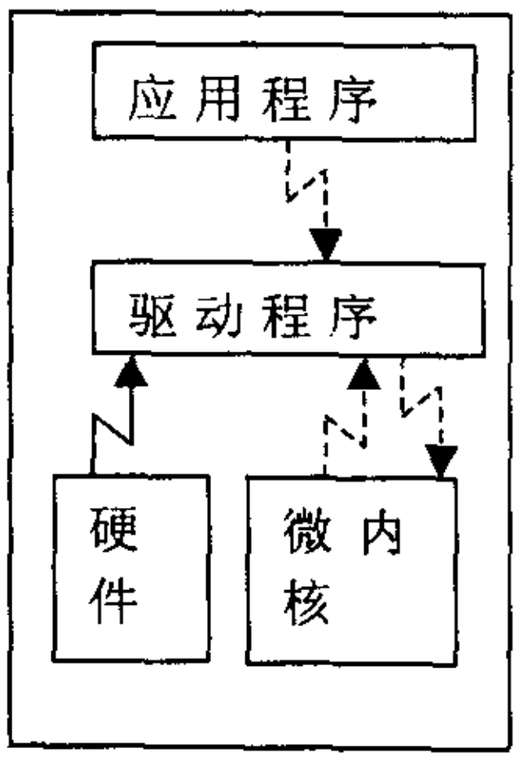

图 2-7 微内核 I/O

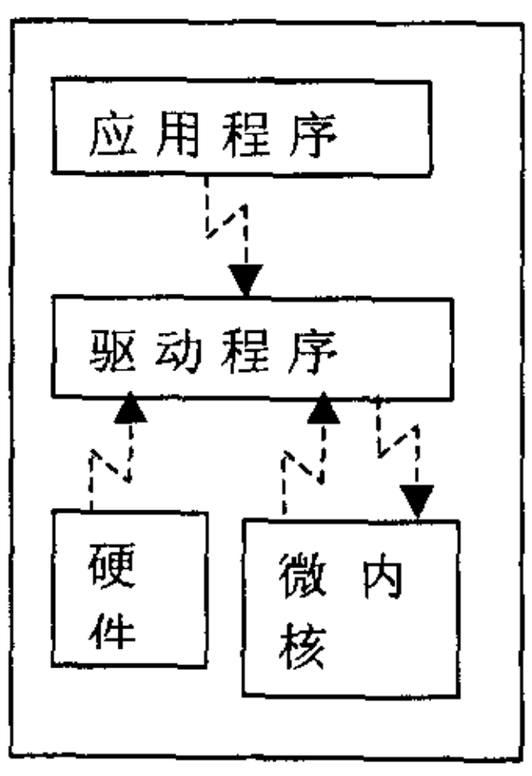

图 2-8 改进 I/O

2.3 小结

本章首先提出了 ARTs-OS 的 I/O 总体结构设计图。然后，在提出 I/O 结构复杂性的评价原则的基础上，具体的分析了 ARTs-OS 的 I/O 结构复杂的原因，并通过有针对性的改变局部连接类型的方法，将 ARTs-OS 的 I/O 结构复杂性降到了相当低的程度。并且，该方法并不仅仅局限于 ARTs-OS 中，可以推广到其它操作系统的结构设计中去。

3 ARTs-OS 中 I/O 体系结构的详细设计

尽管第二章中已经给出了 ARTs-OS 的 I/O 逻辑结构图，但实际上，在该结构中，还有很多重要的内容需要进一步的细化，这客观上要求进行详细的结构设计。这些重要的、需要进行细致地分析和回答的问题有：

1. 需要细化设计的内容是什么？也就是详细设计究竟要设计些什么要素？

2. 这些需要设计的要素可以取那些“合理的值”，也就是它们可以被设计成什么情形？

3. 不同的详细设计方案会对整个 I/O 系统的结构和性能产生什么样的影响？

其实，这些问题都归为一个重要内容：含糊的 I/O 构架的确切含义是什么？本章将回答这些问题，并且，在本章中，还将详尽地分析 ARTs-OS 所有可能的 I/O 构架，并指出它们各自的特点。这种详细设计方法对于设计普通操作系统的 I/O 结构也具有一定的参考意义。

3.1 ARTs-OS 操作系统 I/O 结构详细设计的内容

概括的讲，ARTs-OS 的 I/O 体系结构是指：操作系统 I/O 的核心部件——驱动程序的结构，以及驱动程序与硬件、内核、应用程序部分的连接关系的集合。也就是：一个部件，三个连接件。

驱动程序作为一个部件存在，但它的结构往往不能由自身来决定，事实上，它的结构，是由与它相关联的三个连接件来决定。这就是为什么驱动程序员经常被要求驱动程序必须按照这样或那样的规范来写，但从驱动程序自身却看不出被要求的理由。因为，这些规范实际上就是驱动程序的三个连接件的具体表现形式。下面将具体讨论与驱动程序相关联的三个连接件：驱动与硬件的连接；驱动与内核的连接；驱动与应用程序的连接。

3.1.1 驱动程序与硬件的连接关系

在 ARTs-OS 中，驱动程序与硬件之间主要关心两个问题：驱动程序如何操纵硬件以及如何处理硬件中断。

1. 驱动程序如何操纵硬件

一般驱动程序最终是要操纵物理硬件设备来完成实际的读/写操作(当然也有虚拟驱动程序,但它们也有虚拟设备以供操纵)。具体可分为如下三个问题:

(1) 驱动程序如何读/写端口

读写端口一般有两种方式:In/Out 端口操作和直接访存指令。但无论采用何种方式,都不会对驱动程序的结构产生影响。而事实上,绝大部分的操作系统将 I/O 指令用 HAL(Hardware Abstract Level 硬件抽象层)屏蔽,抽象出来的是统一的 I/O 指令,驱动程序员根本不需要知道它们在实际指令上的差异,更不要说对驱动程序的结构产生影响。

(2) 如何得到完成通知

驱动程序确定物理硬件完成 I/O 操作的方法一般可分为两种:程序控制和中断方式。程序控制是指驱动程序主动周期性地访问硬件中相应的状态标识,以决定物理设备是否完成。中断方式是由硬件通过中断机制,将完成信号主动地传递到操作系统,再由操作系统传递给相应的驱动程序。除了少数简单的单片机操作系统外,几乎所有的操作系统,都无一例外的采用中断方式作为设备完成信号的通知方式。所以,本文将仅讨论中断方式,而忽略程序控制方式。而实际上,程序控制方式是同步硬调用,因此,其结构复杂性也要比中断方式(异步硬调用)小。

(3) 如何传输数据

驱动程序传输数据的方式有两种:程序 I/O(PIO)和 DMA 方式。若采用 PIO 方式,驱动程序同步读/写设备端口,不会对程序结构产生影响。若采用 DMA 方式,由于 DMA 最终还是采用中断机制完成操作,其对结构的影响也就转化为中断机制对结构的影响。

2. 驱动程序如何处理中断

正如第二章中所述,中断机制是对驱动程序的结构复杂性产生重要影响的因素。驱动程序对中断的处理方式将直接影响 I/O 结构的复杂度,并对 I/O 的实时响应性能也有很大的影响。下节中将对此作专门的分析。在实现中,硬件中断最终将通过操作系统的中断派遣机制传递到驱动程序中,所以,将在驱动程序与内核(中断派遣机制)的关系中对其进行详细讨论。

3.1.2 驱动程序与内核的连接关系

1. ARTs-OS 内核的组成

操作系统内核的概念很模糊，不同的操作系统，内核中包含的内容很不一样。如：整体式内核的 Linux，它在内核中几乎包含所有的驱动程序和系统级服务；而在 ARTs-OS 中，内核仅仅包括：进程/线程调度、IPC、最小的内存管理和中断管理机制。在本文中，由于需要分析驱动程序和内核的关系，自然在内核的概念中将不再包括驱动程序。

2. 驱动程序与内核的连接关系

驱动程序与内核的连接关系最终分解为驱动程序与内核各个组成部分之间的连接关系。具体为以下四种连接关系：

(1) 驱动程序与中断派遣机制之间的连接关系：中断处理方式。由于硬件中断是异步事件，因此，按照连接件分类观点，它们之间的连接关系只能分为硬异步连接和软异步连接两种。

(2) 驱动程序与内核调度之间的连接关系：驱动程序的存在方式。驱动程序作为一个部件，当它与内核调度联系时，将产生两种存在方式：一种是作为一个独立调度单位（进程/线程）而存在；另一种是作为共享代码存在，依附于其它调度单位（如：用户进程）而被执行。

(3) 驱动程序与内存管理之间的连接关系：驱动程序的存在位置。内存管理将内存划分为核心、核外两大部分。驱动程序与内存的连接关系将决定它处在内存中的哪一个部分。由此可被划分为：核内驱动和核外驱动两大类。

(4) 驱动程序与 IPC 机制之间的关系：驱动程序的同步与异步。驱动程序的同步与异步可进一步分为两个子问题：

① 驱动程序与应用程序之间的同步与异步。也就是驱动程序阻塞式(同步)和非阻塞式（异步）的系统调用界面。常见的读/写操作是同步方式，但也有操作系统提供异步的调用界面。

② 驱动程序与硬件中断之间的同步与异步。这是指驱动程序完成一个 I/O 请求时，是否采用同步（等待中断），还是异步（与硬件并行操作）完成。

3.1.3 驱动程序与应用程序的连接关系

驱动程序与应用程序的连接关系反映出来的就是驱动程序的

系统 API 调用界面。它对应用程序如何使用驱动程序提供的 I/O 功能有着很大的影响。

但是，驱动程序与应用程序间的连接是受驱动程序与内核结构的连接关系的影响。例如：若驱动程序与内核中内存管理之间的关联为代码共享方式存在，那么该驱动程序必须“借用”应用程序的进程/线程来执行 I/O 功能，而它也必须占用应用程序的地址空间。因此，此类连接将归并到驱动程序与系统内核的连接关系中进行讨论。

在 ARTs-OS 中，由于是采用微内核结构，在该结构下，驱动程序作为一个独立的调度单位运行，那么事实上，它是不会区分一个 I/O 请求是来自核外的用户进程，还是核内的调页请求的（除了优先级上有差别外）。从这个角度上讲，将驱动程序与应用程序之间的关系归并到驱动与内核的关系也是可行的。并且，这种简化并没有减少 I/O 结构分析所应包含的内容。

3.1.4 小结

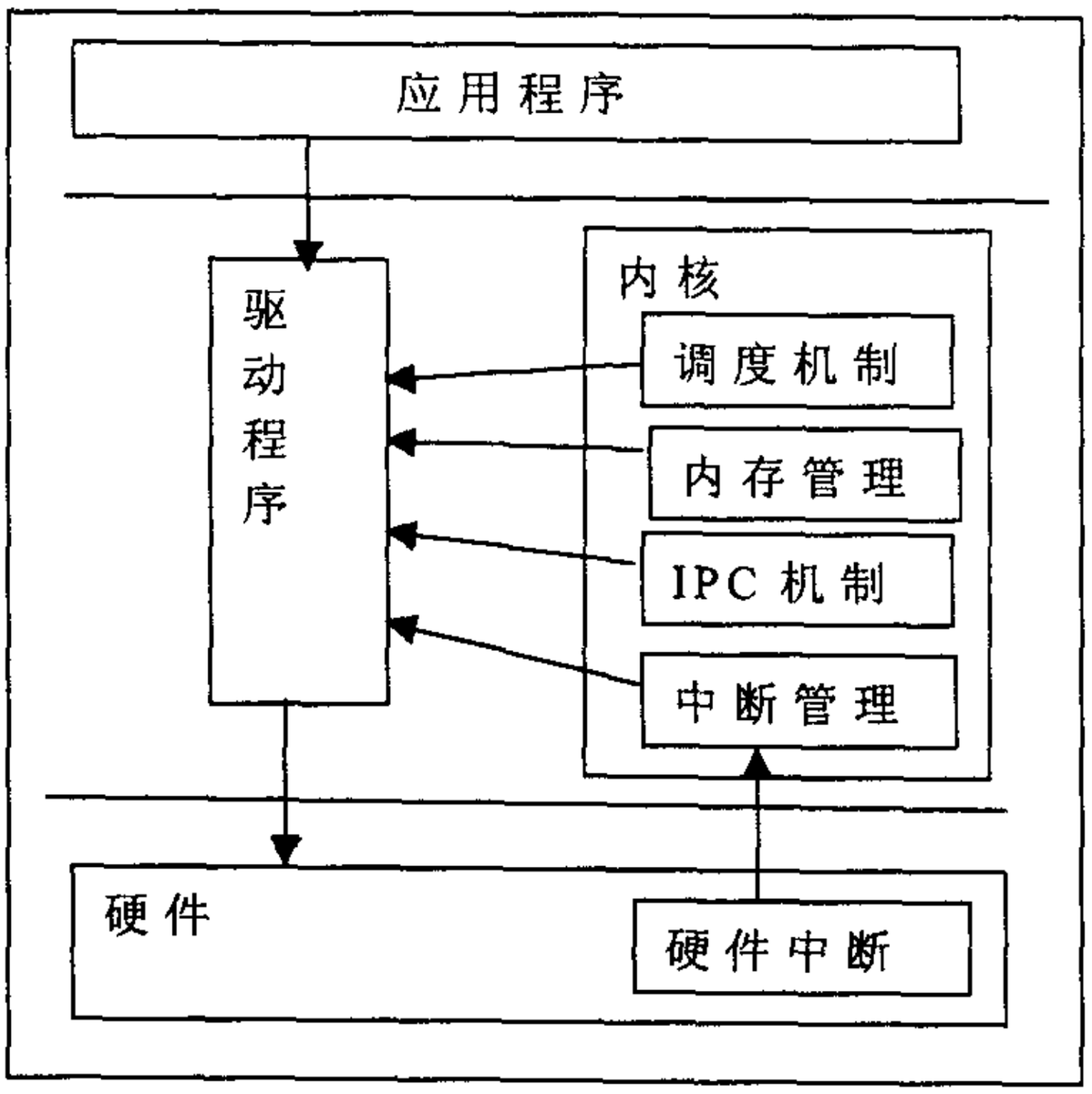

图 3－1ARTs-OS 驱动与内核关系图

图 3-1 完整描述了 ARTs-OS 中驱动程序同与之相关联的各个部件的连接关系。驱动程序与三个部件的连接关系最终集中到驱动程序与内核部件的 4 种连接关系上。即：I/O 与中断管理的关系——中断处理方式；I/O 与调度的关系——驱动存在方式；I/O 与内存管理的关系——驱动存在位置；I/O 与 IPC 的关系——驱动的同步与异步。对这四种连接关系作更详尽地分析将有助于我们认识操作系统的 I/O 体系结构的实质含义。

3.2 ARTs-OS 中 I/O 的中断处理设计

3.2.1 ARTs-OS 的中断处理逻辑结构图

从图 3-2 中我们可以看出：

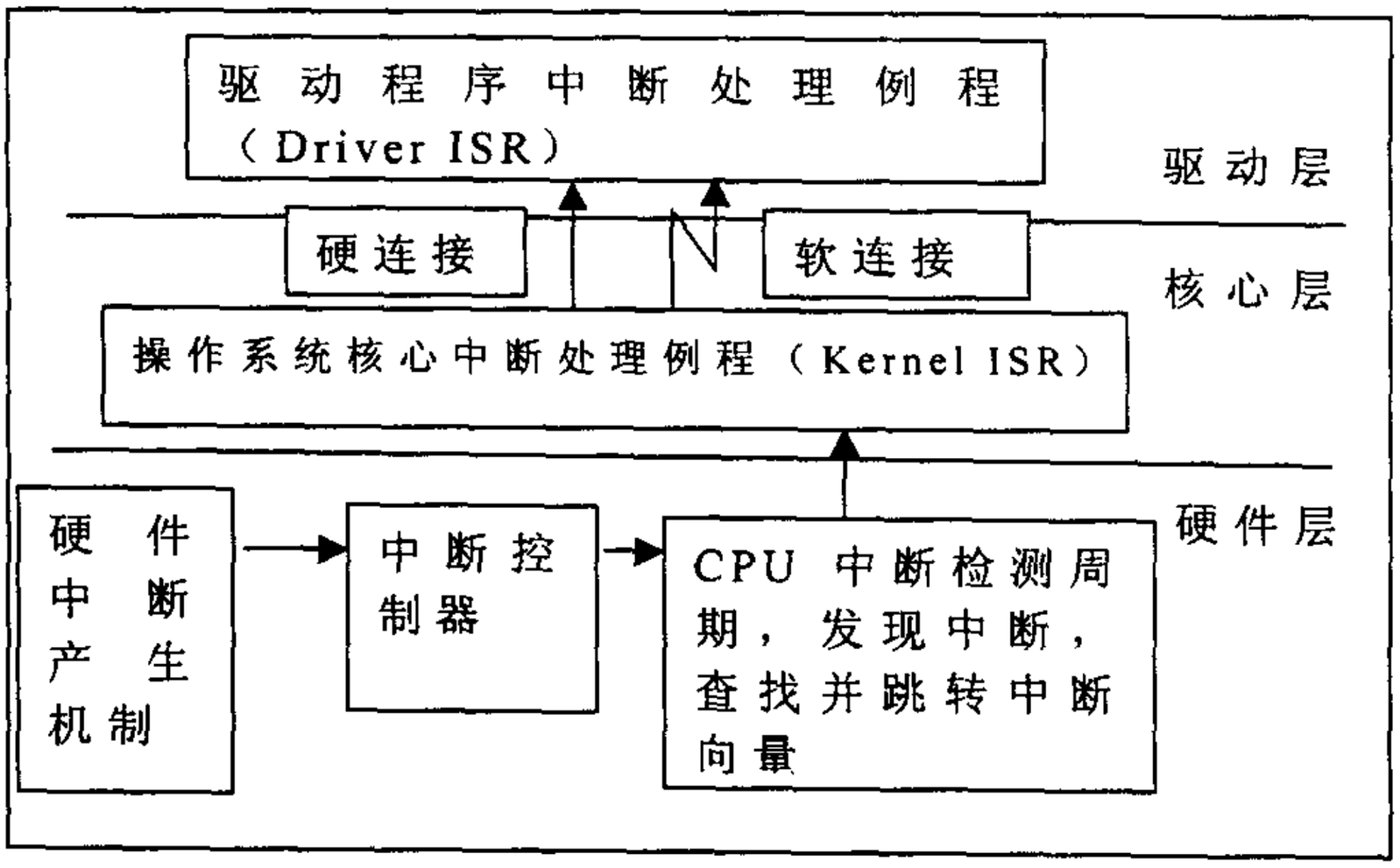

图 3-2 ARTs-OS 的中断处理逻辑结构图

1. 中断信号从设备，经过系统中的中断控制器，到 CPU 查找中断向量表并转化为中断向量，直到启动核心 ISR，这均是硬件完成，是中断处理中不可更改的部分。因此，以下将不再讨论中断的这一部分。

2. 中断一般是特定于设备的。只有驱动程序自己才真正知道应该如何处理该中断。因此，中断信号最终应到达驱动程序并由驱动程序作相应的处理。可能例外的是系统中实时时钟芯片产生

的中断，它一般在系统核心中被处理，但若将时钟芯片抽象为一个时钟设备，则上述依然成立。

3. 在图 3－2 中，驱动层和核心层没有处在同一个层次，是指它们是在不同的逻辑结构层中。但在有些操作系统中，它们可以同在一层中（如：Linux 中就都在核心层）。当然，也可以在不同层中，如：操作系统 ARTs-OS。

3.2.2 ARTs-OS 的中断处理方式

中断处理方式可以看作核心 ISR 与驱动程序 ISR 之间的连接关系。由于我们这里讨论的中断均为异步连接[1]，所以，中断处理方式也就仅有异步硬连接和异步软连接两种。

异步硬连接是指操作系统核心 ISR 直接调用驱动程序 ISR，或者，甚至直接将驱动程序 ISR 的首地址填入中断向量表中，以实现直接由硬件跳转到驱动程序 ISR。异步软连接是指操作系统核心 ISR 将中断信号，以某种方式（消息、信号）发给驱动程序 ISR，然后，当驱动程序检测到该信号后，再进行相应处理。

在 ARTs-OS 的中断处理的详细设计中，为了不同的设计理念，同时存在着两种中断处理方式：中断软连接和中断硬连接。下面将分别详细地介绍在 ARTs-OS 中的这两种中断处理方式。

1. ARTs-OS 中 I/O 中的软中断处理

(1) 实现软中断的原因

在 ARTs-OS 中，最基本的中断处理方式是基于消息机制的软中断处理方式。采用这种该中断处理方式的原因主要有：

① 为使驱动程序结构简洁、清晰。正如第二章中阐述的，采用软中断后，中断事件被“弱化”为一个普通消息（尽管优先级较高）发给驱动程序，在驱动程序的结构中不再为中断作单独的设计，使驱动程序显现出来的是一个统一的界面：接收消息，处理消息的循环。但它的劣势也是非常明显的，那就是：中断消息处理存在不确定的延迟。这一点对于硬实时应用是不可忍受的。

② 在某些场合中，实现硬中断很困难。硬中断除了对驱动程序的结构复杂性有一定程度的影响外，另外一个实际的因素是它实现上的困难。例如：当驱动程序在核外时，实现硬中断将是

[1]这里的中断仅指外部时间触发的中断，而不包括由 INT 指令导致的中断。而后者为同步连接。

一件非常困难的事情，如果不是不可能的话。这也是限制着硬中断在某些简单、软实时事务中应用的一个重要的因素（实现硬中断困难的具体原因，将在硬中断中阐述）。

(2) 软中断的适用范围

① 软中断由于采用消息机制，因此，必须要有消息缓冲区、消息收发机制等一整套消息机制的支持。在 ARTs-OS 中，IPC 机制很好地完成了这一工作，并且，还额外地提供了实时系统中特有的基于优先级的消息机制。

② 驱动程序自身必须利用消息循环来主动检测消息缓冲区。

③ 正因为拥有消息循环，驱动程序必须存在独立的调度单位——进程/线程，来执行检测消息队列的消息循环。因此，驱动程序不能实现为代码共享方式（参见“驱动程序存在方式”小节）。

(3) 软中断的特点

① 驱动程序结构简单、清晰、统一；

② 中断存在着不确定的时延；

③ 实现上简单，无技术困难。

2. ARTs-OS 中 I/O 的硬中断处理

由于软中断的一个重要弱点：不确定的中断时延，在 ARTs-OS 中增加了另外一种硬中断处理方式以适用于硬实时场合。

(1) 异步硬中断连接的适用范围

① 驱动程序 ISR 必须保证硬连接的“时空约束”。也即：时间上，在核心 ISR 调用它时，它必须存在，而不能处于未加载状态或调出内存（若系统支持核心缺页则也可以存在于虚存上，但 IA32 构架不支持这一特性）；在空间上，核心 ISR 必须能够直接调用驱动程序 ISR。驱动程序的 ISR 不能够与核心处在完全独立的地址空间中。（CPU 的过程调用是不支持从一个地址空间中调用到另一个完全独立的地址空间中）。

相对来讲，前者比较容易满足。因为在实时嵌入式系统中，为了保证执行的确定性，很少采用虚存交换技术；即便是在通用操作系统中，也很少将驱动程序调换到虚存上去，而更多的是将驱动程序直接放在不可换出的核心内存上。

而后者对驱动程序位置存在很大影响。由于核心 ISR 处在核心内存中，要想它直接调用驱动程序 ISR，则必须满足：

a. 核心 ISR 和驱动程序 ISR 在同一个地址空间中。即从核心空间可以访问驱动程序所在的地址空间，这是最低的要求。这一

点容易满足，因为，几乎所有的操作系统都把核心空间用共享方式占用着普通应用程序的一部分地址空间。

b. 必须满足核心 ISR 到驱动程序 ISR 的直接可调用。一般 CPU 的“环状保护”机制允许过程调用发生在同级或外层对内层的调用，而禁止内层对外层的调用。为了满足这一点，只有两个方法：

(a) 驱动程序 ISR 和核心 ISR 处在同一个层，也就是：驱动程序 ISR 处在核心。这就是大部分操作系统所做的。当有中断发生时，直接调用相应的驱动程序 ISR，实现硬中断响应。

(b) 绕过 CPU 芯片的保护机制。若驱动程序 ISR 处在核外，唯一的方法是设法绕过保护机制，实现由内核向核外 ISR 的过程调用。这也是实现核外驱动的一个较大的困难（若一定要用硬连接实现中断机制），它需要利用 CPU 保护机制的“漏洞”，来实现这种硬连接。ARTs-OS 的实践证实，至少在 IA32 构架上该种方法是可行的。将在第五章中介绍这一技巧。

② 中断发生时，一般不进行进程/线程切换的。也就是说，可能在另一个进程执行时发生中断，那么这时，驱动程序 ISR 就在该进程的核心堆栈上执行，而不是先切换到它所在的进程，然后再在其核心堆栈和地址空间中执行。因此，在驱动程序 ISR 执行时，是不能对其上下文环境作任何假设的。但是，这种对上下文不作任何假设的要求有时是过于严格的，甚至妨碍驱动程序 ISR 完成其工作。例如：在 ARTs-OS 中，一个典型的驱动程序 ISR 应做如下的工作：

a. 检查硬件的状态标识，确定是否正常完成 I/O 工作；

b. 若正常完成，则从设备上的数据端口读（或写）取数据到某一个缓冲区（驱动程序缓冲区或应用程序缓冲区）；

c. 重置设备，清设备 READY 标识；

d. 设置驱动程序中的某个全局标志位，指示操作完成，唤醒驱动程序。

其中，缓冲区的首地址、驱动程序的全局标志位等往往存在于驱动程序地址空间中。在执行驱动程序 ISR 时，若能保证它的上下文环境是驱动程序所依附的地址空间，则极大地方便了驱动程序 ISR 的编写。它可以象普通的例程一样读/写全局变量。唯一不同的是，这个 ISR 例程，除了被系统核心 ISR 调用外，不会被驱动程序自身的任何其它函数调用。

在实现上，若驱动程序在核心中，则驱动程序的所有全局变量均可以直接访问，不存在上下文环境的问题。但是，若驱动程

序在核外实现，为了便于访问驱动程序的全局变量，可以有如下两种方案：

a. 将驱动程序的缓冲区、全局变量等加载到核心中。它要求特殊的加载机制使驱动程序数据与程序分离。并且，驱动程序自身访问缓冲区、全局变量要通过系统调用界面，变得低效。

b. 在驱动程序 ISR 执行时将上下文切换到相应的驱动程序中。这会造成中断响应延迟稍微加大。

这两种方案均是可能的，在 ARTs-OS 中是采用的后者。

(2) 硬中断处理的特点

① 采用硬连接最大的优点是：中断延迟的确定性。由于是直接从内核 ISR 调用（甚至直接从硬件跳转）到驱动程序 ISR，中断延迟时间是由硬件开销和极少的系统指令组成。所以，它一般很小，这对硬实时应用是非常有利的。

② 正如第二章分析结论，硬连接的中断处理使驱动程序变成了一个双入口，双出口的程序，增加了驱动程序的结构复杂性。

③ 采用硬中断，对驱动程序所处的保护层次有很大的限制。若驱动程序在内核，实现上较为简单；但若驱动程序在核外实现，则实现非常的困难。

④ 硬连接要求驱动程序 ISR 的执行时间应足够短。在硬连接时，驱动程序 ISR 实际上相当于内核执行时间的延伸，若它执行时间过长，将对其它低优先级的中断响应造成很大的影响，也对系统执行时间的确定性有损害。

⑤ 核外驱动的硬连接，对驱动程序 ISR 的执行环境提出了要求。它要求在相应的驱动程序上下文中执行，而核内驱动程序无此要求。

3.3 ARTs-OS 中驱动程序存在方式的设计

3.3.1 ARTs-OS 中驱动程序的存在方式

驱动程序的客户是指使用驱动程序服务功能的进程/线程。它可以为应用进程，这是最多的一种客户。但它也可以为内核线程，如：核内虚存交换请求的发出者。还可以为另一个驱动程序，如：文件系统往往是磁盘驱动程序的驱动客户。但核心的中断调度机制不是驱动程序的客户，它并不需要驱动程序提供的 I/O 服务。

驱动程序要得到执行权利，必须首先在核心的调驱动程序要得到执行权利，必须首先在核心的调度中存在调度的单位。如果驱动程序在启动时就得到自己独占的调度单位，并一直保持到其结束服务，则我们称这种驱动程序存在方式为进程/线程存在。否则，驱动程序提供服务需要依附于其客户的调度单位，则称为共享代码的存在方式。

在实现上，反映在连接件的分类中，就是驱动程序和驱动程序客户之间的连接方式。拥有独立调度单位的驱动程序一般拥有消息队列，因此，驱动程序客户和驱动程序之间采用软连接的方式。若驱动程序没有自己的调度单位，则驱动程序客户和驱动程序之间一般采用硬连接的方式。

在 ARTs-OS 中，驱动程序的存在方式是：拥有独立的调度单位，而不是代码共享的方式。

3.3.2 ARTs-OS 与 Linux 在驱动存在方式上的对比

ARTs-OS 的 I/O 采用的是进程/线程模型，而 Linux 操作系统则采用的是代码共享的方式实现驱动。将它们进行对比，可以清楚的看到各自的优缺点，有利于 I/O 结构的详细设计。

1. 中断处理方式上的比较

在 ARTs-OS 中，可以采用软中断或硬中断两种连接方式，并可以同时存在。而在 Linux 中，只能采用硬中断的方式。因为用代码共享方式时，没有独立的线程检测消息队列中的消息。

2. 程序的可重入要求

在 ARTs-OS 中，不要求驱动代码为可重入代码。而在 Linux 中，由于代码共享，驱动程序在同一时刻可能有多个客户同时请求服务，因此，必须实现代码的可重入。

3. 驱动与驱动客户间的连接方式

在 ARTs-OS 中，驱动与驱动客户间只能为软连接。而在 Linux 中相反，它只能为硬连接。并且，由于代码共享，驱动程序必须存在于所有驱动程序客户的地址空间中。

4. 驱动程序核外、核内的位置要求

在 ARTs-OS 中，驱动程序可以实现在核外或核内。实现核外驱动，虽然也有一些具体的技术问题，但是实现起来还是相对简单些。但在 Linux 中，实现核外驱动则非常的困难。当驱动程序代码存在于核心空间中，并且该核心空间以共享方式占用所有用

户的地址空间（大部分操作系统是这样的），则驱动程序可以为各种驱动程序客户提供服务。但当驱动程序以共享方式存在于用户空间中（非核心空间），则驱动程序很难为核心的驱动客户提供服务（核心调用用户空间的代码受到保护机制的约束，道理同核外硬中断一样）。而且，需要所有驱动客户预留在核外需要共享驱动程序的代码空间，并且位置要相同。

5. 动态加载的实现

在 ARTs-OS 中实现动态加载相对要容易一些。但在 Linux 中实现核内动态加载则相当的复杂。如果是核外的动态加载，几乎是不可能的，因为要修改全部正在运行的应用程序的某些地址空间，那将是非常低效的且容易造成混乱。

6. 执行效率

在 ARTs-OS 中，驱动客户发出 I/O 请求到驱动程序执行返回，需要往返于核内与核外，执行效率要比 Linux 的共享代码方式低。通过“轻量级消息机制”可以缓解这一效率问题。

7. 实现 I/O 排序的复杂性

驱动程序必须有某种机制实现 I/O 请求的排序。I/O 请求的排序是指将 I/O 操作请求的进请求队列的序列转化为物理执行序列的过程。

在 ARTs-OS 中，实现 I/O 排序相对要容易理解的多。它是在检测消息队列的时候，从消息队列中挑选合适的 I/O 请求并执行 I/O 操作。挑选的过程就是 I/O 排序的执行时刻。而在 Linux 中，由于代码共享方式，驱动程序没有自己独立的线程，如何激活自己读取自身的 I/O 请求队列是值得认真考虑的问题。

图 3-3 用生产者－消费者模型分析了 Linux 下的 I/O 排序。生产者协议：驱动程序将 I/O 请求放入 I/O 请求队列中（假设队列足够长）。消费者协议：若设备不忙且 I/O 请求队列不空，采用合适的排序算法取下一个 I/O 请求，启动该请求的物理读写操作。

在 Linux 中，驱动程序有两个激活点：一个是驱动程序客户发出 I/O 请求（生产者激活点）；另一个是硬件产生中断（消费者激活点）。显然，前者是唯一的 I/O 请求生产者；而后者是唯一的 I/O 请求消费者。值得注意的是，在 I/O 请求产生后，该作为生产者的激活点必须立即向消费者的角色转换，即图 3-3 中的斜虚线所示。否则，消费者激活点将无法启动。从中我们可以看出，在没有独立调度单位的情况下，实现 I/O 请求的排队，需要仔细地设计生产者与消费者的协议关系，否则极易产生错误。

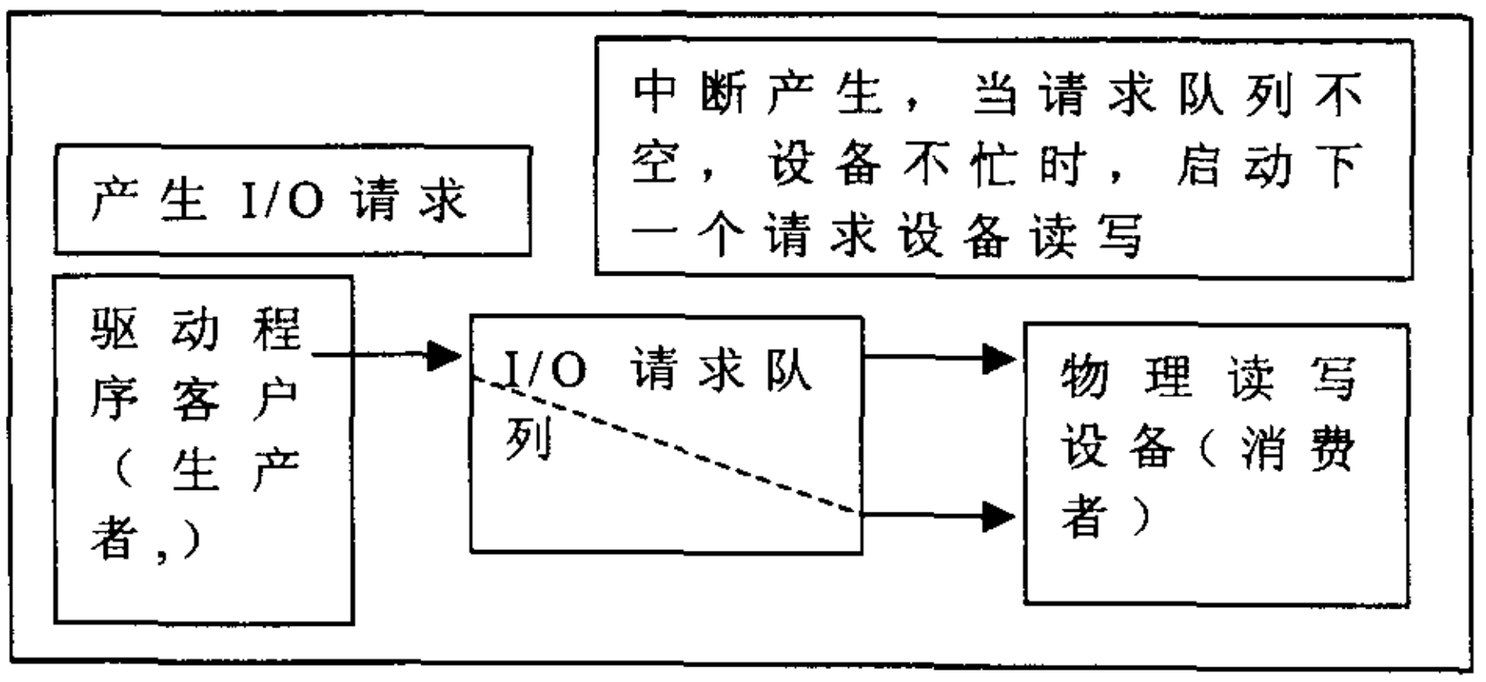

图 3-3 Linux 实现 I/O 请求排序

3.4 ARTs-OS 中驱动程序存在位置的设计

在计算机的保护机制中，操作系统的代码和应用程序的代码处在不同的保护级别。一般“环状保护”模型中，操作系统处于核心——第 0 级，而应用程序处在核外——第 3 级。驱动程序无论作为代码或者进程/线程存在，它都将被加载到某一个层次，也就是驱动程序的存在位置。它反映的是驱动程序和核心内存管理的关系，也是操作系统 I/O 体系结构中一个重要的因素。

在 ARTs-OS 中，驱动程序可以由驱动程序员指定，而被加载到核心或者核外，并且两种存在位置可以同时并存。

3.4.1 ARTs-OS 核内驱动的特点

1. 仅从结构上考虑，实现核内驱动相对简单。这一点可以从上小节的讨论中看出。在 ARTs-OS 中，若实现为核内驱动，则不需要很多特别的处理，且无需绕过保护机制等特别的困难。

2. 核内驱动无需 I/O 特权级的特别处理。一般 I/O 指令为特权指令，在核内缺省为可以直接操作，而若在核外，则需要特别的打开指定的 I/O 指令才能够不出现保护异常错误。

3. 核内驱动调试困难，由于在核内，很多的调试技术不能使用。

4. 核内驱动需要内核预留较大的空间。由于内核的空间一般是静态分配，为了保证驱动程序的正确加载，必然需要更多的内核空间。

5. 核内驱动对系统内核的稳定性有危害。由于驱动程序在核内，所有的操作是不受任何限制的，因此，存在破坏内核的可能。

6. 核内驱动一般需要较长的“I/O步长”。驱动程序在核内，相应的缓冲区也就在核内，数据从物理设备到缓冲区，然后从缓冲区到用户程序空间，而这后一步的拷贝操作一般是不可少的。但在核外驱动中往往实现为较短的“I/O步长”，它可以把数据直接从物理设备传输到应用程序空间。但是，在ARTs-OS中，也可以实现核内驱动的“一步I/O”，但这需要调用特殊的内存管理例程。

3.4.2 ARTs-OS核外驱动的特点

1. 由前面的分析可以得知，若各种连接采用硬连接，在核外实现驱动程序是有很多的困难。特别是硬件中断的处理。

2. 它能够避免昂贵的上下文切换，保护模式的出入，数据流在不同级间的拷贝，而且在结构上，它具有：灵活性，有效性，可移植性等优点。

3. 核外驱动对系统内核的影响很小，对内核而言，它就相当于一个普通的应用程序。

4. 调试方便。

3.5 ARTs-OS中驱动程序的同步与异步设计

驱动程序的同步与异步有两个含义：一是指驱动程序与驱动程序客户之间的调用接口的同步与异步关系，称为外同步；另一个是指驱动程序与物理设备之间的同步和异步关系，称为内同步。

3.5.1 ARTs-OS驱动程序的外同步

外同步就是我们通常说的阻塞式系统调用，所有的操作系统提供同步调用接口，这也是最常用的接口方式。但也有少数的操作系统提供异步调用接口，驱动客户在发出读/写请求后并不阻塞，而是可以继续执行其它代码。当然，到某一个时刻，该客户需要调用专用的同步调用，来同步驱动程序的I/O操作。

由第二章的结构复杂性评价原则，驱动程序采用异步接口会增加应用程序在执行I/O操作时的复杂性，但是，它的优势也很

明显，它可以显著提高应用程序与 I/O 操作的并行度。

正是基于减少应用程序结构复杂性的考虑，在 ARTs-OS 中，驱动程序的外同步目前仅支持同步的调用界面。但是，也可以在不对驱动程序作任何更改的前提下，很容易的提供异步的调用界面，这只需要更改在用户空间的运行时库（Runtime Library）就可以提供异步的 API 调用接口。

3.5.2 ARTs-OS 驱动程序的内同步

当驱动程序向设备发出 I/O 指令后，否等待物理设备的完成（产生中断），则为同步，若不等待，则为异步。显然，同步时，

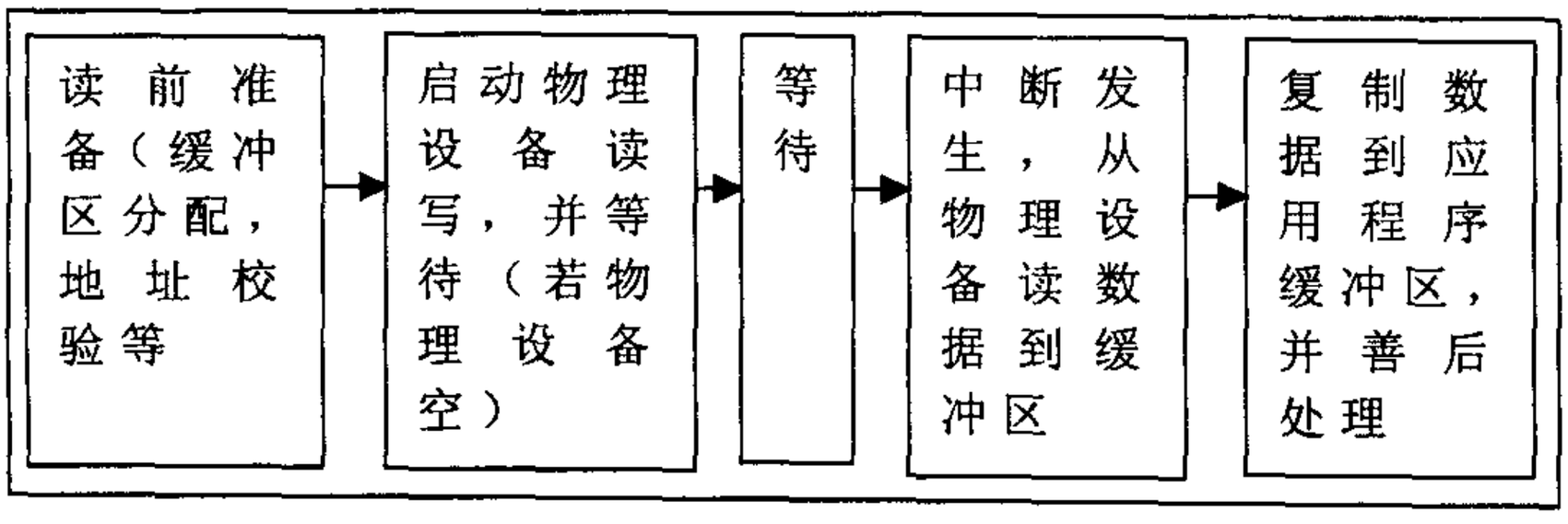

图 3－4： 典型 I/O 请求执行过程

表 3－1 I/O 操作各步骤占用系统资源表

工序	1. 读前准备	2. 启动物理设备	3. 等待中断	4. 读数据	5. 善后
占用资源	CPU	CPU + I/O 设备	I/O 设备	CPU + I/O 设备（DMA）	CPU

驱动程序阻塞在等待物理设备的中断上，也就不能处理其它可能不需要物理设备参与的 I/O 操作，即：不能使多个 I/O 操作并行。

如图 3-4 中，在 ARTs-OS 中，一个典型的 I/O 操作被分为了 5 个“工序”，各“工序”占用系统资源的情况如表 3-1：

由表 3-1 中可以看出：在步骤 3 中，并不需要 CPU，可以让 CPU 进行其它的处理工作，而在步骤 1，5 中恰好仅要 CPU 而不需要 I/O 设备，因此，多个 I/O 请求可以并行地操作。但若驱动程序为内同步，则它只能“忙等”。

同样，出于驱动程序结构清晰上的考虑，目前，在 ARTs-OS 中仅支持内同步。若要支持内部的并行异步操作，则远比实现外部的异步复杂得多，它需要对驱动程序的结构作相当大的调整。但从具体实现上考虑，未来准备用多线程模拟，效果是一样，但结构上要简单些。

3.6 ARTs-OS 中 I/O 体系结构的综合分析

3.6.1 ARTs-OS 中 I/O 详细设计的综合

在前四节中，详尽地分析了 ARTs-OS 中驱动程序的每一种连接关系的所有可能情况。由于这四种连接关系是正交的，因此，操作系统所有可能的 I/O 结构就只能为它们的组合情况。本节将综合分析这些组合情况。

从表 3－2 中我们可以总结如下规律：

1. 若实现为代码方式，一般采用中断硬连接，而若实现为进程/线程方式，则采用中断软连接。

2. 微内核操作系统，一般实现为进程/线程方式，可以体现出微内核的优势，ARTs-OS 就是这样。而整体式内核操作系统通常采用的是代码共享方式。

3. 核外很少实现为代码共享方式，因为从实现的角度上，需要为每一个进程预留核外的地址空间，这是比较麻烦的。况且，动态加载驱动程序将修改所有的应用程序空间，显然是不太实际的。所以，几乎很少见到这种实现方式。

4. 核外的硬中断实现比较困难。

表 3－2 操作系统 I/O 结构组合表

编号	中断	代码	核内	异步	描述	例子
0	Y	Y	Y	Y	中断硬连接，核内代码，I/O 为异步操作	Linux
1	Y	Y	Y	N	核内代码方式，硬中断，同步 I/O 操作	
2，3	Y	Y	N	Y N	核外代码共享，硬中断，同步/异步 I/O 操作	实现非常困难
4	Y	N	Y	Y	核内线程，硬中断，异步 I/O	
5	Y	N	Y	N	核内线程，硬中断，同步 I/O	Topsy,Windows 2000 等
6	Y	N	N	Y	核外线程，硬中断，异步 I/O	核外实现硬中断困难
7	Y	N	N	N	核外线程，硬中断，同步 I/O	ARTs-OS，QNX 等嵌入式操作系统
8	N	Y	Y	Y	核内代码，软中断，异步 I/O	无。核内很容易实现硬中断
9	N	Y	Y	N	核内代码，软中断，同步 I/O	无
A	N	Y	N	Y	核外代码，软中断，异步 I/O	无。核外共享代码很难实现
B	N	Y	N	N	核外代码，软中断，同步 I/O	无
C	N	N	Y	Y	核内线程，软中断，异步 I/O	
D	N	N	Y	N	核内线程，软中断，同步 I/O	ARTs-OS 早期版本
E	N	N	N	Y	核外线程，软中断，异步 I/O	
F	N	N	N	N	核外线程，软中断，同步 I/O	ARTs-OS 也支持

3.6.2 ARTs-OS 的 I/O 体系结构特点

ARTs-OS 是本论文的原型操作系统。它的 I/O 体系结构就是在对 I/O 结构进行仔细分析的基础上进行设计的。它的主要技术特点有：

1. 基于微内核构架。由于我们整个操作系统的体系结构定位在微内核结构上，I/O 部分也不例外，采用了较先进的微内核设计。

2. 支持动态加载。驱动程序的加载级别为动态运行时可加载。加载驱动程序就如同加载一个普通的应用程序一样简单。操作系统不用重新启动。甚至可以实现为自动加载/卸载模式，即：驱动程序的加载和卸载是系统自动完成，对用户而言，所有逻辑设备均可用，它的加载/卸载对用户程序透明。

3. 核内/核外驱动。驱动程序可以在核外执行，因此，所有核外驱动的优势它均具有。同时也可以在核内运行，如：键盘、显示器驱动一般在核内（为内核调试必备的）。两种驱动存在方式可以同时存在。

4. 进程/线程模型。驱动程序执行时，是作为独立的进程/线程运行，只是它比普通应用程序的优先级高一点而已。

5. 中断硬连接。即使在核外，也可以透明地使用硬连接中断，如同在核内实现一样。并且，是在驱动程序的上下文中执行中断 ISR，所以，驱动 ISR 可以象一个普通函数一样编写、编译，也可以访问驱动程序的全局变量，这方便了驱动 ISR 的编程。同时，我们也支持中断的软连接。

6. 驱动程序内部为同步 I/O，外部也为同步 I/O 操作。为了简化驱动程序编写，目前在驱动程序中只支持同步的 I/O 操作。但在应用程序接口中，可以为外部的同步或异步 I/O 操作。

7. 可自定义的 I/O 调度策略。驱动程序员可以方便地编写自己的 I/O 调度策略。

8. 支持优先级逆转的防止。

9. 支持减少 I/O 步长的操作，即：“一步 I/O”。可以将数据直接在 I/O 设备和应用程序空间中进行交换，而不需要在驱动程序缓冲区和应用程序空间之间拷贝数据。

10. 支持实时磁盘技术。

3.6.3 ARTs-OS 中驱动程序的结构

1. ARTs-OS 中驱动程序框架

(1) 定义驱动程序数据结构

```
driver{
name; //驱动程序名称;
(major, minor); //(主, 次)设备号, 是驱动的唯一标识;
(interrupt_no, interrupt_ISR);// (中断号, 中断 ISR);
(read_function,  write_function,  open_function,
close_function , etc); //操作函数;
} Driver;
```

(2) 登记驱动程序

```
register_driver( &Driver);
```

(3) 启动消息循环

```
while ( receive_message( &message )) {
case open: call open _function; break;
case close: call close _function; break;
case read: call read_function; break;
case write: call write _function; break;
etc...
}
```

2. ARTs-OS 读/写函数框架

```
read_function (physical_add, length, buffer_add)
{
    (1) 校验地址的合法性;
    check(physical_add, length, buffer_add);
    (2) 启动物理设备读/写
    port_io (.....)  //启动设备
    (3) 等待特定的中断消息
    receive_message(中断消息); //这里表明该驱动程序内部
    为同步 I/O; 阻塞等待;
    (4) 从设备读数据
    port_io(....); //从设备和缓冲区交换数据, 交换直接到达
    驱动程序客户空间。即: 一步 I/O.
    (5) 复制数据
```

无。//此过程在我们驱动程序中没有。

}

3. ARTs-OS 中断 ISR 函数框架

interrupt_ISR() {

send_message(中断到达); //唤醒阻塞在读/写操作上的驱动程序;

}

//中断 ISR 做的工作很少，仅仅发送一条消息，当然，也可以将读/写操作中的步骤 4 写在中断中。无论怎样，一定要保证硬中断 ISR 的简洁，快速。因为它是系统执行时间的一部分。

4 实时磁盘调度算法的评价

4.1 现有实时磁盘调度算法分析的缺陷

随着多媒体和嵌入式领域的发展，实时磁盘调度算法得到广泛的重视和深入的研究。各种新的实时磁盘调度算法层出不穷。但是“如何评价一个实时磁盘调度算法的优劣？”这一问题却很少得到应有的重视和细致的研究。常见的方法是建立一个模拟运行平台进行模拟试验，得出相应的数据再进行分析[37]。当然，这是最“实际”的检验方式。但是，该方法存在如下的弊端：

1. 建立实际平台，工作量相当大，需要耗费大量的工作时间；若建立模拟的运行平台，虽然工作量较小，但是由于其基于排队论，很多重要的参数，如：磁盘请求到达率，请求特性（位置，长度等），要想与实际情况相符，并非易事。

2. 在实际测试中，由于参量过多，环境影响因素很多，得出的结果很难做到精确。

3. 更重要的是，这种评价实质上是一种宏观上的，粗糙的比较，它不能刻画在某些关键的调度点上，带有转折意味的调度行为将导致完全不同的调度结果。而只有那样才能对改进算法有实质性的帮助。

如果存在一种模型，它能在调度算法作微观调度决定时，评估你的调度决定对未来的影响，并且，告诉你决定的“理性”程度，势必对改进算法起到良好的作用。本文就是基于这种想法，提出了一种评价模型，它能够从磁盘效率和实时可满足性两个方面，对每一个调度方案比较在当前所有可能方案中的优劣。本文最后，还应用该模型设计了若干试验，对几种常见的调度算法作了比较，得出了一些不寻常的结论。

磁盘调度算法有很多。对于传统的非实时磁盘调度算法有：FCFS，SCAN，SST 等。但它们并不能满足实时性要求。也已有大量的文献研究过实时磁盘算法并提出的相关算法。如：EDF[38]，SCAN-EDF，FD-SCAN[39]，DM-SCAN[40]，enlarged-MSG[41]等。

理论上可以证明，若每个磁盘的 I/O 请求的执行时间是固定的（还有其它几个假设），EDF 算法是满足实时性最好的算法[42]。但遗憾的是，这个假设并不成立。

影响实时磁盘调度算法性能的主要因素有两个：第一，I/O请求在磁盘上的物理位置，一般认为，电梯算法和SST算法对于提高磁盘的吞吐量是有效的；第二，实时I/O请求的截止期限制。实时磁盘调度算法的关键困难在于，这两个因素是不相干的。如何将两者“理智”的结合起来，是衡量算法好坏的关键。

过分考虑吞吐量，可能会使某些I/O请求等待时间过长而变得“危险”；过分考虑截止期，会使磁头产生很多不必要的寻道，效率下降。在某些比较“困难”的局面下，“聪明”的算法，会从并不多的选择中，找到可以全部完成的路径；或者，在较“安全”的情况下，适当提高“危险”程度，却可以极大地提高吞吐量。这种微观的决策，往往就决定了一个算法的宏观表现。

4.2 实时磁盘调度算法的评价模型

4.2.1 实时磁盘调度模型

模型为：一个单台磁盘、单头、读系统、一个读请求队列、无缓冲。也即：每一个读请求必须从磁盘上获得。每次读且仅读一个磁道。磁盘的磁道编号为 track = [1, Tracks]。p0 为磁头起始位置，显然：p0∈[1, Tracks]。

定义 1 I/O 请求 T = (track, arriveTime, deadLine) 三元组。其中：

1. track∈[1, Tracks];
2. arriveTime 为该请求到达读请求队列的时刻；
3. deadLine 为该请求的截止期。

定义 2 读请求队列 Queue 为一个集合。记为 Queue = { t_i | $t_i \in T$ }。等待队列长度为 m = #(Queue) 。

定义 3 一个调度方案 p_k 为 Queue 中所有元素的一个 m 元组。也即：

$p_k = (t_1, t_2, \cdots t_m)$ 其中：$t_i \in$ Queue, i = 1..m, 且 $t_{i1} = t_{i2}$ 当且仅当 i1 = i2。

定义 4 可选调度集 P 为 Queue 的全排列的集合。也即：

$P = \{ p \mid p = (t_1, t_2, \cdots t_m), t_i \in$ Queue, i = 1..m; $t_{i1} = t_{i2}$ 当且仅当 i1 = i2 }。

可调度的总数为 n = # (P) = m !。

定义 5 在调度方案 p_k 中，t_i 的寻道数记为 $Seek(t_i / p_k)$，定义为：

$$Seek(t_i / p_k) = \begin{cases} |t_i.track - t_{i-1}.track| & 当\ i \neq 1; \\ |t_i.track - p0| & 当\ i = 1; \end{cases}$$

注意：现代磁盘的寻道数和寻道时间并不成正比[43,44]，更准确的衡量应该用寻道时间，但为了讨论的方便，这里仍然用寻道数代替寻道时间。

在调度方案 p_k 下，总的寻道时间为：

$$Seek(p_k) = \sum_{i=1}^{m} Seek(t_i / p_k)。$$

定义 6 在调度方案 p_k 中，t_i 的完成时刻记为 $finish(t_i / p_k)$，定义为：

$$finish(t_i / p_k) = \begin{cases} finish(t_{i-1} / p_k) + Diskfactor * sqrt(Seek(t_i / p_k)) + DiskConst & 当\ i \neq 1; \\ Diskfactor * sqrt(Seek(t_i / p_k)) + DiskConst & 当\ i = 1; \end{cases}$$

其中：

Diskconst：为一常数，表示磁盘的旋转时间和传输时间。

Diskfactor：为一常数，表示磁盘的寻道时间因子。

定义 7 在调度方案 p_k 中，t_i 的紧迫度记为 $\Phi(t_i / p_k)$，定义为：

$$\Phi(t_i / p_k) = \frac{finish(t_i / p_k) - t_i.arriveTime}{t_i.deadLine - t_i.arriveTime}$$

由 Φ 的定义我们可以知道：

$0 < \Phi < 1$ 时，在调度方案 p_k 中，I/O 请求 t_i 可以提前完成；

$\Phi = 1$ 时，在调度方案 p_k 中，I/O 请求 t_i 可以刚好按时完成；

$\Phi > 1$ 时，在调度方案 p_k 中，I/O 请求 t_i 不能按时完成，需要超时。

定义 8 调度方案 p_k 的紧迫度 $\Phi(p_k)$ 定义为：

$\Phi(p_k) = MAX\{\Phi(t_i / p_k)\}$，其中：$i = 1..m$.

定义 9 调度方案 p_k 的不可满足集 Miss（p_k）定义为：

Miss（p_k）= { t_i | t_i ∈ Queue，i = 1..m，且 Φ（t_i / p_k）>1}

调度方案 p_k 的不可满足 I/O 请求个数为 MissNumber(p_k) = # Miss（p_k）。

定义 10 不可满足调度集记为 P_{unsat}，定义为：

P_{unsat} = { p_k | p_k ∈ P，# Miss（p_k）> 0 }

定义 11 等待队列 Queue 的难度，记为 Difficulty (Queue)，定义为：

Difficulty (Queue) = # P_{unsat} / # P ；

由定义可知，Difficulty (Queue) ∈ [0，1]；且：当 Difficulty (Queue) = 0 时，任何调度方案均可按时完成当前队列中的所有 I/O 请求；当 Difficulty (Queue) = 1 时，任何调度方案均不能按时完成当前队列中的所有 I/O 请求；若是随机的挑选调度方案，则，Difficulty (Queue)越大，能够全部按时完成请求的可能性就越小。反之相反。

4.2.2 调度评价模型的建立

定义 12 二元关系 $\geqslant_{prior}$（p_1，p_2）定义为：

$\geqslant_{prior}$（p_1，p_2）= {（p_1，p_2）| p_1 ∈ P，p_2 ∈ P，Φ（p_1）≤ Φ（p_2）且 Seek(p_1) ≤Seek(p_2)}

可以证明：$\geqslant_{prior}$ 关系是偏序关系（自反，反对称，可传递）。

定义 13 理性调度集 P_r 定义为：

P_r = { p_1 | p_1 ∈ P，∀p_2 ∈ P，$p_1 \neq p_2$，（p_2，p_1）∉ $\geqslant_{prior}$ }

若作一个二维图形，以寻道数 Seek 为 X 轴，紧迫度 Φ 为 Y 轴，将调度方案 p_k 在图形中表示出来，则，若在该图形中，将理性调度集 P_r 中的元素，按照 Φ 值的大小排列，并依次用线段连接起来，则称该曲线为理性调度边界。

定义 14 实时理性调度集 P_{rt} 定义为：

P_{rt} = { p_1 | p_1 ∈ P_r，Miss（p_1）= Min{Miss(P_2)}，P_2 ∈ P_r }

同样：在 Φ－Seek 图中，将实时理性调度集 P_{rt} 中的元素，

按照Φ值的大小排列，并依次用线段连接起来，则称该曲线为实时理性调度边界。

容易证明如下的一些结论：

1. 不在理性调度集 P_r 中的元素，必在理性调度边界的右上方。（证略）

2. 理性调度边界的每一条线段的斜率必为负。（证略）

3. 实时理性调度边界中的调度方案之间在 $\geqslant_{prior}$ 关系意义下，是不可比较的。

4. 在实时理性调度边界中，最高点是紧迫度最大，寻道数最少的方案。最低点是紧迫度最小，寻道数最大的方案。

定义 15 一个调度方案 p_k 的优于集 $Prior(p_k)$定义为：

$$Prior(p_k) = \{ p_l \mid p_l \in P,\ p_k \in P,\ p_l \neq p_k,\ (p_k, p_l) \in \geqslant_{prior} \}$$

同理，一个调度方案 p_k 的劣于集 $Inferior(p_k)$定义为：

$$Inferior(p_k) = \{ p_l \mid p_l \in P,\ p_k \in P,\ p_l \neq p_k,\ (p_l, p_k) \in \geqslant_{prior} \}$$

如下结论是很容易证明的：

1. $\#Prior(p_k) + \#Inferior(p_k) \leqslant \#(P)$；

2. $\#Inferior(p_r) = 0$，$\#Inferior(p_{rt}) = 0$；

3. $Prior(p_k)$中的元素在Φ－Seek图中，在 p_k 的右上方（第一象限）；

4. $Inferior(p_k)$中的元素在Φ－Seek图中，在 p_k 的左下方（第三象限）。

若优化目标定义为：在不可满足实时性的请求数最小的前提下，使得寻道数最小。则有如下结论：

定理 1 最优方案均在实时理性调度 P_{rt} 内。

证明：

（反证法）设存在一个最优的调度方案 p_1，且 p_1 不属于 P_{rt}。则由 P_{rt} 的定义可知，在 P_{rt} 必中存在一个 p_2 使得，$\Phi(p_2) \leqslant \Phi(p_1)$ 且 $Seek(p_2) \leqslant Seek(p_1)$，且 $\Phi(p_2) = \Phi(p_1)$ 和 $Seek(p_2) = Seek(p_1)$不同时成立，否则，p_1 属于 P_{rt}。而这 p_2 的存在与 p_1 为最优方案矛盾。

定理 2 在最优方案中，减少寻道数必然导致紧迫度增加。

同上道理，可用反证法证明。

我们可以这样理解如上定理：如果我们设计的算法都足够的好，那么它会命中实时理性边界。但好到这种程度之后（全部命中），如果我们想进一步减少寻道数，那么，必然结果是丧失一部分实时性能。

4.3 评价模型的模拟试验

4.3.1 试验环境

基准值：

Tracks: 1000

Diskfactor: 0.6 ms （寻道因子）

Diskconst: 15.0 ms （旋转时间和传输时间）

$$Access(n) = Diskfactor * \sqrt{n} + Diskconst$$

其中：n 为需要的寻道数。

deadLine = arriveTime + averageAccesstime + slcakTime

其中： arriveTime = 0 （为了简化起见）;

averageAccesstime = 26ms

slcakTime $\in$ [min_slack, max_slack]（至于 min_slack 和 max_slack 将作为负载的调整参数。）

样本点数：m

样本产生方式：

磁头位置为[1,Tracks]上的均匀分布；

slcakTime 为[min_slack, max_slack]上的均匀分布。

比较的算法：

采用了三种典型的算法进行比较：

1. SST：一种倾向于提高吞吐量的有效算法。

2. EDF：最简单的考虑实时因素的算法，若不是不满足一个重要假设，将是可证明的实时性最优的算法。

3. SCAN-EDF：混合 EDF 和电梯算法的一个磁盘调度算法。

本实验全部在 matlab6.1 上进行。

4.3.2 试验结果

1. ϕ — Seek 分布图

图 4-1 的目的是分析各种调度算法在所有可行的调度方案中的分布情况，得出一些大概的结论。

这是在 m = 5; min_slack=50, max_slack=150 参数下一个样本点的结果。

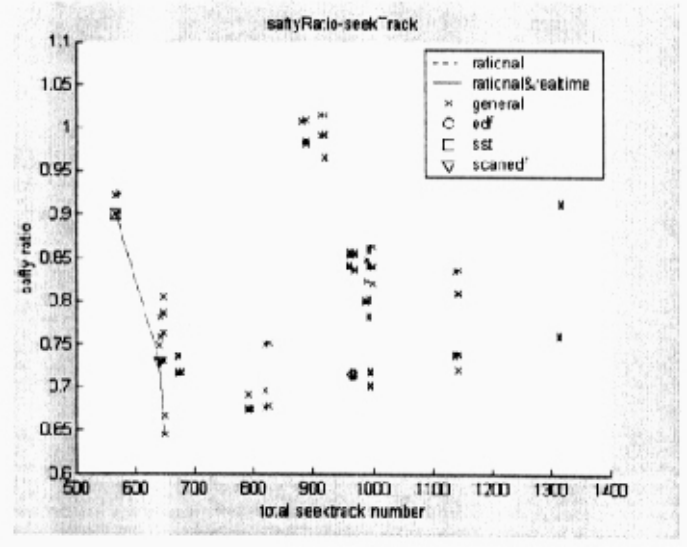

图 4-1 φ — Seek 分布图

从图 4-1 中我们可以看出，正如我们的想象，一般而言，SST 算法的寻道数一般最小，而 EDF 的寻道数一般较大，而 SCAN-EDF 介于两者之间。紧迫度恰好相反。但有一点值得注意的是，EDF 算法的结果离理性边界很远，而 SST 和 SCAN-EDF 算法都离理性边界较近。这不是一个特例，下面将用统计数据说明这一点。

2. 命中实时理性调度边界次数统计图

图 4－2 是在 m = 5 下 3000 个样本点的统计结果：

从图 4－2 中我们可以看出：

(1) SST 和 SCAN-EDF 算法命中实时理性边界的比例明显要高于 EDF 算法。而 SST 和 SCAN-EDF 之间没有明显的差别。

(2) EDF 算法命中 RTB 边界的比例随着格局难度的增加而有所增加。

3. 不可满足调度比例图

从图 4－3 中我们可以看出：

(1) 一般而言，SST 的不可满足调度的比例要高于 SCAN-EDF，

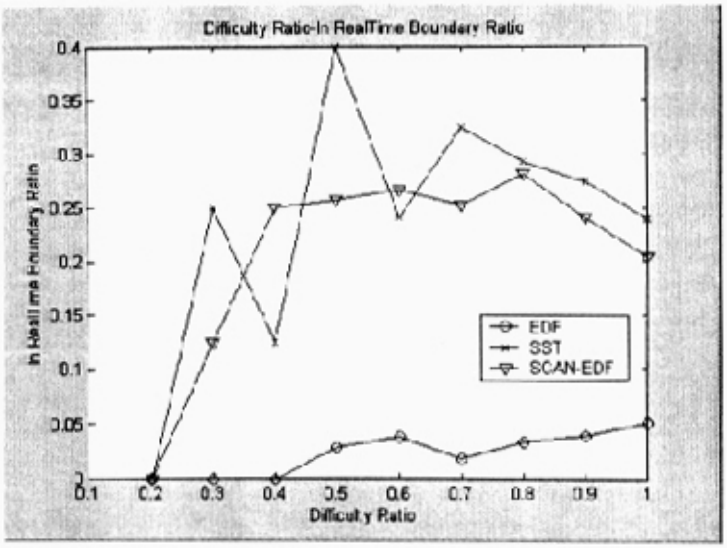

图 4－2 命中实时理性调度边界次数统计图

而 SCAN-EDF 要高于 EDF。

(2) 随着局面难度的增加，它们三者调度失败的比例都会上升。

(3) 当局面的难度较小时（Difficulty (Queue) < 0.5），它们都很少调度失败；在局面非常困难的情况下(Difficulty (Queue) > 0.95)，从可满足上评价，EDF 算法甚至比 SST 和 SCAN-EDF 算法还要差。

4. 寻道数比例和局面难度关系图

从图 4－4 中可以看出：

(1) SST 算法的相对寻道数最小，SCAN-EDF 其次，而 EDF 最大。

(2) EDF 的寻道数，在平均寻道数附件波动。而 SCAN-EDF 在 0.65，SST 在 0.55 附近波动。

(3) 相对寻道数并不随着局面的难度增加而增加。

5. 试验结论

1. 尽管 EDF 算法不满足实时性调度的一个重要的前提假设，也不是最佳的实时调度算法，但从实际的效果来看，当问题的难度在 0.5-0.9 之间时，在可满足实时性能方面，它仍然是三者中

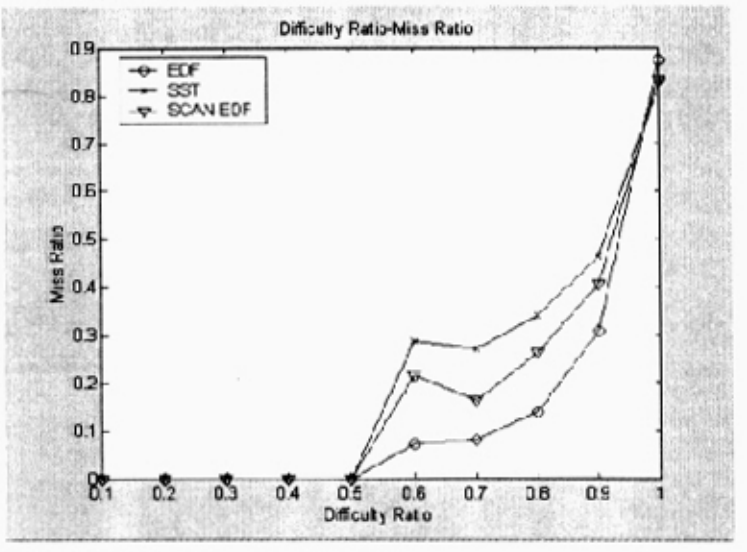

图 4－3 不可满足调度比例图

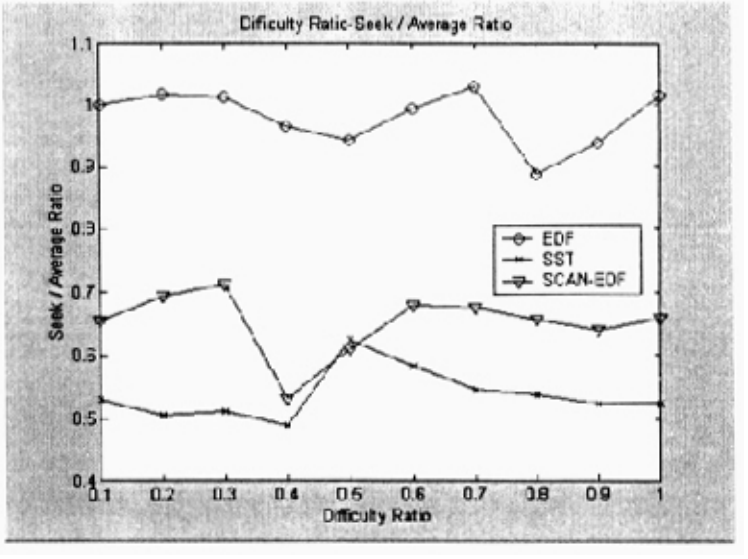

图 4－4 寻道数比例和局面难度关系图

最好。但当问题很难时(难度大于 0.9)，它相反不如电梯一类的

算法。所以，在难度很大的情况下，用电梯一类算法可能是更明智的选择。

2. SCAN-EDF 算法的初衷可能就是为了平衡 EDF 的实时性和电梯算法的高效率（寻道数少）。从统计数据上来看，SCAN-EDF 的寻道数和失败调度数均落在它们之间。因此，这个算法的设计应该是成功的。

3. 从命中实时理性边界的比例上讲，SST 和 SCAN-EDF 要远高于 EDF。这说明 SST 或类似的电梯算法是比较靠近理性边界的，也就是说，作为一个以效率为主的算法，它们是比较成功的。而 EDF 算法则不然，它往往离边界较远（就象图 1 中显示的那样），作为一个以提高实时可满足性为目标的算法，它是不成功的，因为，存在太多的算法比它要好（从实时和效率两个方面都是）。SST 一类算法的目标是命中理性边界的上顶点，它较好的达到这一点；EDF 的目标是命中理性边界的下顶点，而它却没有很好的达到。尽管 EDF 的实时性能要好于 SST，但事实上，拿 EDF 和 SST 比较实时性能是不公平的。EDF 算法不是一个实时性能好的算法，这里有很大的改进余地。

4.4 结论

传统实时调度算法性能评价是建立实际运行平台，得出宏观的运行数据，进行统计分析。本文中认为，宏观的调度结果是由一个个微观的调度方案决定的，因此，从每一个具体的调度局面入手分析当前决策，在所有可能选择中的“理性”程度，同时，对这种评价方式也作了一个模型，并应用该模型设计试验，具体地比较了三种典型的磁盘调度算法。得出的一些重要结论有：

(1) 如果我们设计的算法都足够的好(命中实时理性边界)，那么，我们提高调度效率（减少寻道数）的必然结果是以丧失一部分安全性作为代价。

(2) EDF 算法不是一个实时性能好的算法，有很大的改进余地，尽管它的实时性能在三个算法中是最好的。

5 ARTs-OS 中 I/O 的实现技术

本章将从实现的角度，对 ARTs-OS 的 I/O 结构中设计和实现的关键技术作一定介绍。具体将介绍两个技术难点：驱动程序的动态加载技术和核外驱动技术。

5.1 驱动程序的动态加载技术

常见的驱动程序加载是源代码级加载或者是在操作系统启动时的静态加载。驱动程序的动态加载技术作为操作系统 I/O 设计中的一个高级特性，很少有操作系统能完全实现。因为，实现驱动程序的动态加载有其特定的困难。本节将讨论这一个技术，并分析实现动态加载困难的原因，并用实际的操作系统（Linux 2.2 和 ARTs-OS）分析几种可能的实现方案。

5.1.1 动态加载驱动程序的困难

在动态加载时，驱动程序和操作系统是分开编译的，因此，一个重要的困难是驱动程序对操作系统存在的外部引用问题[45]。这是指：由于驱动程序在核内（在绝大部分的操作系统中，驱动程序都是在核内，若在核外，情况将更加复杂），它往往需要调用系统内核的服务例程，如：kmalloc。若是源代码级的加载，这个外部引用可以在编译、连接阶段解决。若是在操作系统启动时，驱动程序一同加载，也可以在加载时找到定位，解决外部引用问题。在操作系统加载完毕后，这些外部引用信息丢失，就不能解决以后要加载的其它驱动程序的外部引用了。

5.1.2 Linux2.2 动态加载的实现方法

很明显，一种很自然的方法就是保存并维护所有系统内核的符号表。符号表中的表项就是抽象为（符号名，地址）的二元组。在动态加载一个驱动程序时，解决该驱动程序的所有对系统内核的外部引用，并且，由于驱动程序本身也是系统内核的一部分，也要将它自身要导出的符号加在该符号表中。以解决依靠该驱动

程序的其它驱动程序的外部引用。Linux2.2 就是采用的这样一种方法。它在系统中维护着一张内核模块表，在用 insmod 将模块读入虚存中时，利用内核中的导出符号解决模块对内核进程的外部引用问题，它是通过在内存中对模块映象进行修补，并将驱动程序加到内核模块表的表尾。

这种解决方案的缺点是：需要额外地维护内核模块表；该表也占用了宝贵的内核空间；对驱动程序内存映象的修改需要对可执行映像文件有较深入的了解。它的优势在于：一旦动态加载完成，驱动程序与内核之间不需要其它间接的转换，系统可以“全速”运行。因此，对系统的执行效率有利。

5.1.3 ARTs-OS 的解决方案

在 ARTs-OS 中，采用的是另外一种完全不同的解决方案。我们将驱动程序实现为进程/线程模型，在该模型下，IPC 机制为其唯一的接口，驱动程序对内核的调用抽象为一系列的消息发送和接收。因此，避免了代码方式下的外部引用的问题。同时，加载一个驱动程序和加载一个普通应用程序没有本质上的差别，除了优先级较高、可以 I/O 操作等一些特权外，所以，实现起来相对 Linux 要容易得多。

驱动程序和内核之间的直接调用（硬连接）被消息机制（软连接）代替，它们通过事先约定的消息协议进行通讯。这相当于用消息协议对内核中的系统调用进行了一次封装，从而避免了驱动程序的外部引用问题。

这种实现方案的优势是：在结构上，驱动程序符合微内核的构架，不需要对驱动程序的二进制映象进行修改。其缺点是：由于采用的是软连接方式，实时性能差。且（消息名，调用地址）的转换，在每一次系统调用的时候都会发生，而不象 Linux 解决方案中，仅仅在加载时进行解析，因此，执行同一个系统调用，这种解决方案需要的时间更长。

5.2 核外驱动技术

普通操作系统的驱动程序位于核内[46,47]。核外驱动的一个重要问题是硬件中断处理。若采用软中断连接方式，在实现上比较简单，但对系统的实时响应性能有影响。但如何实现核外的硬连

接驱动，在技术上是有一定的难度。本节将简要地介绍核外驱动的关键技术——核外硬中断的实现技巧。

5.2.1 核外硬中断实现的困难

核外硬中断是指，当硬件产生中断时，系统核心 ISR 保存现场，然后跳转到核外驱动程序 ISR 并执行，执行完后，恢复现场的过程。

在这个过程中，首先要解决的是，系统如何从核心 ISR 跳转到核外的驱动程序 ISR。若该 ISR 的代码段在核内，由于处于同一个保护层次中，则可以直接的调用。但若驱动在核外，一般保护机制是不允许这样的调用的。例如：在 I386 体系中，核心处在第 0 层，应用程序处在第 3 层，过程调用指令 CALL，只允许从第 3 层到第 0 层的调用（更准确的是：允许>=i 层的对 i 层的调用），而对从第 0 层到第 3 层的调用则产生保护异常。如何越过该保护机制，是第一个困难。

其次，驱动程序 ISR 执行完毕后，跳转到什么地方去是另外一个值得仔细考虑的问题。比较好的方法是：返回到系统内核 ISR 调用驱动程序 ISR 的地方。但是，它在实现上是困难的。因为，一般的过程调用，是通过 CALL 和 RETURN 指令，以及返回地址的堆栈保存这种“过程调用/返回”协议，实现自动地返回到调用点（的下一条指令）。然而，当驱动程序在核外时，它们使用的根本就不是同一个堆栈，核内 ISR 使用 0 层堆栈，核外驱动 ISR 使用被中断应用程序的地址空间中的 3 层堆栈。如何实现这种切换返回需要仔细地考虑。

第三，如何处理驱动程序 ISR 中对驱动程序中全局变量（例如：驱动程序缓冲区）的访问。在一般的函数中，是不存在这样的问题，但在驱动程序 ISR 中，这将成为一个很重要的问题。一般的函数是由该函数所在地址空间的其它函数所调用，当能够执行到该指令时，由 CPU 的进程/线程调度机制已经将该进程的地址空间恢复，自然，普通函数根本就不知道进程的地址空间在 CPU 上被不断地切换这一事实。但对于中断响应函数 ISR 就不是这样。驱动 ISR 是由操作系统内核（具体就是：内核的中断 ISR）调用，而内核中断 ISR 被调用的时机与操作系统自身的运行是异步的，也就是，在任何时候都有可能发生硬件中断。因此，有可能在另外一个应用程序在运行时，发生硬件中断，从而调用驱动程序

ISR，如果不加特别的处理驱动程序 ISR 访问的全局变量将是另外一个应用程序空间中的地址。

5.2.2 核外硬中断的实现技巧

如何同时解决上述三个问题，将是实现核外硬中断技术的关键。本节将介绍 ARTs-OS 的核外硬中断在实现上的一些技巧[2]。

1. 越过保护机制的跳转的实现

由于 CALL/JMP 类指令有保护机制的约束，只能由外向内跳转，但 RET 指令恰好相反，只能由内向外跳，因此，一个很“常用”的技术的就是采用 RET 指令实现由内向外的“调用”。具体如下：

(1) 在堆栈上压入需要调用的核外驱动 ISR 代码的首地址 CS:IP。在保护模式下 CS 为段选择子

(2) 执行 RET。硬件将从堆栈上弹出 CS:IP；安全检查（显然可以通过）；然后执行之。

2. 核外驱动返回的实现

核外驱动 ISR 执行完后，要返回到内核 ISR 的调用处。实现这一点，不能采用常规的返回技术。而应采用“堆栈执行”的技巧，即：在堆栈上压入汇编代码，然后利用返回指令执行该代码，实现重返内核。具体如下：

(1) 在调用驱动 ISR 之前，应作一定准备工作：

(2) 找到核外驱动程序 ISR 将使用的堆栈；

(3) 在堆栈中压入代码，该代码主要是实现 INT n 的系统调用，重返内核，还包括平衡堆栈的代码；

(4) 将代码的首地址压入堆栈，作为返回地址；

(5) 建立好过程调用的“调用帧”的前半段后，用 RET 指令进入该驱动程序 ISR。

当执行到驱动程序 ISR 的 RET 语句时（注：该 RET 编译后为一个段内近调用，因为编译器并不知道该函数会被系统“回调”，而是把它当作一个普通的函数进行编译），由于返回地址为堆栈上事先压入代码的首地址，执行该代码：在平衡堆栈后，用 INT 指令重返内核。

[2] 注：核外硬中断的设计和实现是由中断处理小组完成，而非 I/O 资源管理小组完成，在此描述仅仅出于对 I/O 结构讨论完整性的考虑．代码细节请参阅相应小组文档．

3. 驱动程序地址空间的恢复

为了方便驱动程序 ISR 访问驱动程序空间中的全局变量，应当在进入核外驱动 ISR 之前恢复该驱动程序的地址空间。这类似于进程切换，首先将该驱动程序强制性切换到运行态（恢复其地址空间等等），然后执行其中的 ISR。具体细节略。

显然，采用该方法，相当于在每一次硬件中断时都会有至少一次的进程切换。因此，它的中断延迟相对于核内中断要大一些，但与核外软中断相比，它的延迟还是很小的。其它软中断的实现方案都要求在该驱动程序被调度时，才能够处理该中断，显然，中断延迟将更大，且不确定。

6 结束语

随着对实时应用需求的增加，实时嵌入式操作系统的研制得到越来越多的重视。全世界现有实时嵌入式操作系统约一千多种，但它们的输入/输出（I/O）部分却非常的凌乱。

现有绝大部分对 I/O 结构的研究都集中在对既定操作系统的 I/O 结构的描述，如：大量充斥的对 Linux 的 I/O 结构分析；或者为某些特性的实现细节，如：核外驱动、动态加载等高级特性的实现技术。却很少有从整体上分析并回答：为什么操作系统的 I/O 结构如此凌乱？我们究竟该如何分析一个陌生操作系统的 I/O 结构？如果是设计 I/O 系统，哪些结构是可行的，哪些是不可行的，我们在结构设计上的自由度的极限是什么？

通过分析、设计和实现 ARTs-OS 的 I/O 系统，对这些问题作了明确的回答。做的比较有新意的工作主要有以下几点：

1. 在对 ARTs-OS 的 I/O 结构的分析中，发现结构复杂的原因是连接种类的多样化，特别是在有硬件中断的情形下的异步连接。并找到降低 ARTs-OS 结构复杂性的方法。

2. 设计全新或者分析陌生的 I/O 结构需要也仅需要分析其四大要素。即：中断处理、存在方式、存在位置、同步与异步。并且，在对 ARTs-OS 的 I/O 详细设计中发现，这四要素间并非是完全的自由组合，其中一些组合是不可能的。

在实时磁盘调度算法方面，论文可谓汗牛充栋，但如何衡量算法的好坏，往往只能用试验（或模拟）的方法。文中提出了一种新的模型，并用其得出如下结论：

1. 理论证明：在理性调度下，实时性和效率是矛盾的。

2. 试验证实：EDF 算法不是一个实时性能好的算法。

在实现 ARTs-OS 的动态加载、核外驱动等高级特性中，有一些特殊的技巧。文中对实现中遇到的关键技术难点作了实现技术的对比分析和具体实现细节上的介绍。

但是，也留下了一些领域需要更透彻地研究。如：操作系统 I/O 结构的形式化及形式推导，它将使我们对 I/O 结构的本质有更深入地了解；文中的试验数据表明，足够好的实时磁盘调度算法还远没有找到，在此领域将一定会有重大的突破。

致谢

很高兴能够写到这里，因为这意味着几十天没日没夜乱了作息规律的日子快到头了；但也有一丝惆怅，因为离和一起工作、学习过的老师、同学分手的时刻也不远了。细细想来，在整个长达 2 年的操作系统项目中，乃至整个 3 年的研究生生活中，给我指导、帮助的人真的很多。

刘云生老师我想应当是我们所有人都应当首先感谢的，没有他在外不顾酷暑的奔波，就没有这个项目，也就没有我，包括我们组所有的成员，对操作系统深入的思考。但更重要的是他认真的工作态度，记得在住院的时候，手里还抱着一大摞的论文；还有他严谨治学态度与当前浮躁的学术界形成的反差，使我唯一想说的是：他是一位值得尊敬的人。张文彬老师是一位和蔼可亲的导师，向她求助或讨论问题，绝对是一件愉快的事情。她给我时间上的安排，常常给我一种醍醐灌顶的感觉。现在想起来，若不是这些提醒，我现在开始写的可能是序论。龙老师的幽默让我每次和他谈话时都忍不住想笑，不过前几天对数据结构的讨论，才使我看到他出题原来是如此的“凶狠”。曹老师是绝对不容忘记的，她要我提到实验室去的几篮子水果，我还吃了好多。李国徽作为年轻的副教授一直是我们敬仰的对象，哎，只可惜我们太笨，不过和他讨论问题，让你感觉就是不一样。刘南辉管理技巧，使实验室的条件和氛围绝对是一流的，我都有一种做不出东西有点内疚的感觉，幸好我都完成的差不多了。许贵平老师一天到晚催我们的进度，但他在技术上给我的几次建议，至少让我少走几个月的弯路。徐长醒是位让人佩服的博士，和他讨论问题你可以深入到芯片里面去。还有给了我很多帮助的师兄、师姐们，他们是：潘琳、杨进才、陈基雄、夏家莉、秦飙、李蓉蓉、党德鹏、廖国琼。

不应当忘记的还有众多我的“战友们”，他们是：吴飞，一个聪明的帅哥；梁爽，你可以在未来的计算机领域听到她的名字；陈怡，一个聪明圆脸的漂亮女孩；徐媛媛，她的英语好的让你不敢相信；郭元苏，绝对的小帅哥；何冰，她冷静、缜密的思维，很难想象出自一个小女孩子的脑袋；李少华，他的编程水平绝对的一流，他负责的中断代码让我感到：廉颇老已；朱晓，年级的 QUAKE 高手，曾经的白云版主，不知道他还有哪些“邪门左道”。

焦金良，是否还在为去香港城市大学读博士要交培养费想不通——不是回归了吗？付蔚，未来一定能做一个学生喜爱的好老师；李红娇，你宏观的思考使我找工作的时候看清了很多的问题；张小芳，我觉得好老师就是你这个样子。还有好多给过我帮助的同学不能忘记，他们有：李琳、吴苗、赵洪武、陈龑等等。

还有老同学郭建伟，他的活动能力常使我自卑。在广州的黄坚、谢伟，上海的王光华，佛山的汝百乐，新疆的周哲……．他们给了我很多的鼓励和帮助。还有我们寝室的李龙、孔艳广、刘会志，他们可能也受够了我的破机器的轰鸣声，天天盼着我早点写完。还有隔壁的黄正波、吴建华、张华、邓家义、刘伟、蒋屹新、欧阳凯、周健、何中华、赵正龙、崔永全，可能还有很多我一时记不起来的同学，他们给了我太多的帮助，认识他们是我一生的财富。

感谢我的父母！

参考文献

[1] William Stalling. Operating Systems: Internals and Design Principles (Fourth Edition). 魏迎梅,王涌. 北京：电子工业出版社，2001. 354~424

[2] Andrew S. Tanenbaum, Albert S. Woodhull. Operating Systems: Design and Implementation (second edition). 王鹏,尤晋元,朱鹏 et al. 北京：电子工业出版社，1999. 115~227

[3] Ray-I Chang, Wei-Kuan Shih, Ruei-Chuan Chang. Multimedia Real-Time Disk Scheduling by Hybrid Local/Global Seek-Optimizing Approaches. in:Parallel and Distributed Systems, 2000. Seventh International Conference on 2000. 323~330

[4] Abbott, Robert, Hector Garcia-Molina. Scheduling Real-Time Transactions: a Performance Evaluation.in: Proceedings of the 14th VLDB Conference. 1998.1~12

[5] Coffman, E.G., M. Hofri. On the Expected Performance of Scanning Disks. SIAM Journal of Computing, 1982,11(1):60~70

[6] 秦啸,庞丽萍,韩宗芬 et al. 非固定双头镜像磁盘实时调度算法的研究.软件学报, 1999,10(9): 996~1002

[7] J.C. Brustoloni, P. Steenkiste. Effects of Buffering Sernantics on I/O Performance. In: Proc. Usenix 2nd Symp. Operating Systems Design and Implementation(OSDI'96), Usenix Assoc, Berkeley, Calif.:1996,227~291

[8] 张丽芬. Mach3.0 的面向对象的设备管理. 北京理工大学学报, 1998,18(6):727~731

[9] 张丽芬. Mach3.0 面向对象 I/O 系统.计算机工程与科学, 1999, 21(6):58~61

[10]称华瑛. Mach3.0 核心的分析. 计算机研究与发展, 1994, 31(9): 1~6

[11]孙凝晖. Mach 的 I/O 系统. 计算机研究与发展, 1994, 31(9):30~35

[12]Hitoshi Araki, Shin Futangami, Kaoru Nitoh. A non-stop updating technique for device driver programs on the IROS platform.in: Communications, 1995. ICC'95 Seattle, 'Gateway to Globalization', 1995 IEEE. International Conference on, Volume:1,

1995. 88~92
[13]K. Noguchi, T. Ohrui, I. Kogiku. Modeling of Communiction Control Software for Advanced Switching Systems. In:TENCON'92(1992). 654~658
[14]Lambert Schaelicke: Architectural Support for User-Level I/O. Ph.D. Dissertation, University of Utah, 2001.
[15]M.A.Blumrich. Protected, User-level DMA for the SHRIMP Multicomputer. In: Proc. 2nd Int'l Symp. High Performance Computer Architecture (HPCA-2). IEEE CS Press. Los Alamitos. Calif.. 1996. 154~165
[16]T.Von Eicken. U-Net: A User-level Network Interface for Parallel and Distributed Computing. in:Proc. 15th ACM Symp. Operating Systems Principles(SOSP-15). ACM Press. New York. 1995.40~53
[17]D. Engler, M. Kasshoek, J. O'Toole Jr. Exokernel: An Operating System Architecture for Application-Level Resource Management. Proc. of ACM SIGOPS, CO, 1995. 34~51
[18]Liedtke, J. Toward Real Microkernels. Communications of the ACM. 1996: 70~77
[19]William Stalling. Operating Systems: Internals and Design Principles (Fourth Edition). 魏迎梅 王涌. 北京：电子工业出版社，2001. 336~351
[20]朱一凡，刘云生. 面向关键任务实时嵌入式操作系统设计技术研究. 仪器仪表标准化与计量，2000(6): 9~12
[21]刘云生,卢炎生,王道忠. 实时数据库系统（RTDBS）及其特征. 华中理工大学学报,1994(6):66~70
[22]刘云生,易岚,余利平. 一个实时数据模型. 小型微型计算机系统，2000(5):549~552
[23]王振宇.在实时系统中消除优先级反向. 深圳大学学报（理工版），1995(12): 24~31
[24]Sadegh Davari. Sources of Unbounded Priority Inversions in Real-time Systems and a Comparative study of Possible Solutions. ACM Operating System Review, 1992, 26(2):110~119
[25]Mary Shaw, David Garlan. Software Architecture: perspectives on an emerging discipline. Prentice Hall, Inc., 1996. 1~18
[26]Baragry, J., Reed, K. Why is it so hard to define software

architecture? In:Software Engineering Conference.1998.Asia Pacific.28~36

[27]Manuel Barrio, Pablo de la Fuente. Software Architecture: Object vs. Process Approach. In: IEEE 1997.Computer Science Society, 1997. Proceedings, XVII International Conference of the Chilean, 1997. 9~15

[28]H.L. Lutfiyya, M.A. Bauer. An Experience Report on Architecture Development. In:Computer Software and Applications Conference. 1996. COMPSAC'96. Proceedings of 20th International, 1996. 378~383

[29]Timothy J. Popp. Software architecture development for product line software. In:Digital Avionics Systems Conference, 1999. Proceedings, 18th. Volume:B.6-6 Vol.2, 1999.. 9.D.4.1~9.D.4.7 Vol.2

[30]Robert J.Allen: A Formal Approach to Software Architecture. Ph.D. Dissertation. CMU, 1997.

[31]Garlan D. Monroe R, Wile D. ACME: an architectural interconnection language. Technical Report. CMU-CS-95-219. 1995.

[32]张家晨，冯铁，陈伟 et al. 基于主动连接件的软件体系结构及其描述方法. 软件学报,2000,11(8):1047~1052

[33]Shaw M.Comparing. Architectural Design Styles. IEEE Software,1995,12(11):27~41

[34]Erich Gamma, Richard Helm, Ralph Johnson et al. Design Patterns: Elements for Resuable Object-Oriented Software. Addison Wesley Longman, Inc. 1995. 232~237

[35]黎伟建. 领域特定的软件体系结构. 中山大学学报(自然科学版）. 1998(6): 9~12

[36]严昌浩，刘云生，张文彬. 软件连接件的分类及其应用研究. 计算机科学，2002. 6.

[37]Dr. Muhammad Younus Javed, Mr. Ihsan Ullah Khan. Simulation and Performance Comparison of Four Disk Scheduling Algorithms. IEEE 2000(2): 10~15

[38]Robert A, Garcia Molina H. Scheduling real-time transactions with disk resident data. In: Proc. 15th. VLDB Conference. 1989. 256~264

[39]Robert A, Garcia-Molina H. Scheduling I/O Request with Deadlines: a Performance Evaluation. In:IEEE Real-time Systems Symposium. 1990. 113~124

[40]Chang,R.L., Shin,W.K., Chang,R.C.. Deadline modification-Scan with maximum scannable-groups for multimedia real-time disk scheduling. In: Proc., IEEE RTSS. 1998. 40~49

[41]Hsung-Pin Chang, Ray-I Chang. Enlarged-Maximum-Scannable-Groups for Real-time Disk Scheduling in a multimedia system. In: COMPSAC 2000. The 24th Annual International. 2000. 383~388

[42]C.L.Liu, James W. Layland. Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment. ACM. 1973, 20(1):46~61.

[43]Bitton, D. J. Gray. Disk shadowing. In: Proc. 14th. VLDB Conference. 331~338.

[44]Seltzer, Margo, P.Chen. J. Disk Scheduling Revisited. In:Proceedings of USENIX. Winter90.313~312.

[45]David A, Rusling. Linux Programming White Papers. 朱珂.北京：机械工业出版社, 2000. 107~113

[46]Ritchie D.M. The Evolution of the UNIX Time-sharing System. AT&T Bell Laboratories Technical Journal, 1984, 63(8), Part 2: 1577~1594

[47]Maurice J. Bach: The Design of the UNIX Operating System. 陈葆钰. 北京:机械工业出版社, 2000. 241~272

附录 1 攻读硕士学位期间发表的学术论文

[1]严昌浩，刘云生，张文彬．软件连接件的分类及其应用研究．计算机科学，2002.6.（署名单位：华中科技大学）