摘要

基于 Linux 的实时操作系统，由于开放源代码，以及 Linux 系统的强大功能和稳定性，日益受到人们的欢迎。但是 Linux 系统本身是一个分时操作系统，所以必须对其进行实时性改造。RTAI 是目前比较成功的 Linux 实时化方案之一，但是它只支持一种基于静态优先级的调度算法，限制了其应用的范围，因此可以改进其调度机制来扩展 RTAI 的应用。

本课题以 Linux 操作系统为研究对象，在 RTAI 基础上设计并实现了一种通用的实时调度框架。通过在 RTAI 实时内核中添加一个实时调度选择模块，然后对实时内核的相关数据结构进行改进，并添加相关的实时调度算法，使得该选择器可以根据不同的属性值来选择需要的调度策略，改变了 RTAI 在实时调度策略上的单一性。文中论述了 Linux 进程调度机制，分析了 Linux 系统实时性不佳的主要原因，归纳了几种提高 Linux 实时性的方法，并对 Red-Linux 的实时调度框架进行分析与研究，从而形成了本文的设计思想。RM 算法和 EDF 算法是最经典的实时调度算法，但在实际应用过程还存在瞬间过载等问题，文中给出了相应的解决方法。最后通过实验结果的分析，验证了通用调度框架和算法设计的正确性和有效性。

关键词： Linux　实时调度　RM 算法　EDF 算法

Abstract

Real-time operating system based on Linux, because of its open source, as well as the strong capability and stability, has being welcomed by people day after day. However, Linux is a time-sharing operating system, it must be improved for real-time. RTAI is one of the most successful improved Linux system of real-time, but it only supports the scheduling algorithm based on static priority, which limits the scope of its application, so that it can be improved its dispatching mechanism to expand the application of RTAI.

Linux operating system is the topic of this article, and a common framework for real-time scheduling is designed and realized based on RTAI. First, a real-time scheduling choosing module is added to the real-time kernel of RTAI, and then the data structure of real-time core is improved. And add the related real-time scheduling algorithm to the common framework, so that the choosing module can select the scheduling strategy needed by the different attribute value, which changes the single scheduling strategy of RTAI. The article analyses Linux kernel source code, discusses the process of scheduling mechanism detailed, figures the main reasons for the poor real-time Linux, and sums up several methods to improve Linux for real-time. The real-time scheduling framework of Red-Linux is analyzed and researched, which forms the design idea of this article. RM algorithm and EDF algorithm is the most classic real-time scheduling algorithm, but in practical application process still exist, such as instant overload problem, and this article solves it. Finally, the results of experiment verifies the correctness and effectiveness of the design of the common scheduling framework and algorithm.

Keyword: Linux real-time dispatching RM scheduling algorithm
EDF scheduling algorithm

目录

第一章 绪论

目前，嵌入式系统正在重塑人们的生活、工作和娱乐方式。嵌入式系统产生了无数的种类，每类都有各自的特点。例如，当今驾驶的汽车就嵌入了智能计算机芯片，它使汽车更轻快、更干净、更容易驾驶。电话系统依靠集成在一起的多个硬件和软件将全世界的人们联系在一起。即使在私人住宅中，也充满了智能家用电器和围绕嵌入式系统建造的集成系统，这些都极大的丰富了人们的生活。上述的这些系统就是通常所说的嵌入式计算机系统，代表了一类为特殊用途而设计的专用计算机系统。根据对外部事件的实时性要求，产生了一类特别的嵌入式系统，称之为嵌入式实时系统。可以这样认为：一个实时系统必须要及时地完成任务和提交服务[1]。实时系统的实例包括数字控制、指挥系统、信号处理和电信系统等，这些系统每天都在为人们提供重要的服务。从上面的实例可以看出，这些应用必须提供及时、可靠地提供有价值的服务，一旦系统工作不正常，将产生非常严重的后果。因此，这些系统应用运行的基础——实时操作系统(Real-Time Operating System，简称 RTOS)就显得格外重要。

1.1 研究背景和意义

在操作系统领域，实时操作系统属于很特别的一类，我们常见的操作系统都是分时系统，最为经典的分时系统是 UNIX 操作系统系列，它们广泛地应用于研究，教育和商业领域。

实时操作系统由于其应用的领域不同，对其概念的理解也有所不同。但通过对各种不同的实时操作系统的研究，可将其定义为[2]：具有通用操作系统的基本功能，并保证任务对外界异步信号及时响应的操作系统。实时操作系统的基本特性是确定性，它的设计要求是在确定的时间界限到来前完成用户任务。

随着计算机的发展，实时操作系统也经过了一个从简单到复杂，从低级到高级的发展过程。早期的实时应用没有实时操作系统作为软件平台，程序员针对确定的目标系统编写应用，代码直接运行在裸机上，系统的资源由应用程序自己管理，多个任务采用轮转法运行。利用这种方法设计的系统代码重用率低，设计和调试的工作量大，但效率较高[3]。

实时操作系统及其应用开发平台的出现，使得应用程序的开发者可以基于开发平台，利用实时操作系统提供的系统调用，完成具体应用程序的编制和调试，

极大压缩了开发的周期和费用。随着应用复杂度的提高，对实时系统的实时性要求也越来越高。目前，有些商用的实时操作系统达到了这方面的要求，但是其价格昂贵，不开放源代码，用户难以根据需求的变化度身定制地剪裁修改。Linux 以其开放源代码、功能强大、可靠、稳定等优势，成为仅次于 Windows 的操作系统。因此国内外很多大学和研究机构开始基于 Linux 构建实时操作系统，但是由于种种原因，几种改进的实时化方案都存在着各自的优缺点。RTAI 是一种成功的改进 Linux 操作系统的方案，改造后的硬实时系统具有良好的性能和移植性，但是应用范围有所限制，可以通过改进其调度机制来扩展其应用。

1.2 嵌入式实时 Linux 及国内外研究概况

在未来的嵌入式计算系统无处不在的时代，需要一个价格低廉、性能成熟、功能全面、便于移植的嵌入式操作系统，而嵌入式 Linux 具有开放源代码、使用成本低、高可靠性、强大的网络功能、具有功能强大的开发工具、多平台支持和大量的说明文档等等优点。正是因为 Linux 有如上优势，使得 Linux 在嵌入式领域获得了广泛的应用，并在嵌入式操作系统市场上所占的份额逐渐扩大，基于 Linux 开发的嵌入式产品也越来越多。但是普通 Linux 是一种通用的分时操作系统，其在调度算法、定时机制、中断处理、不可抢占内核以及虚拟内存管理等问题上无法满足实时应用的要求，所以嵌入式 Linux 还不能直接应用于实时系统中。因此为了有效利用这个资源，越来越多的厂商及技术开发人员热衷于改善 Linux 的实时性，使其能够满足实时应用的需求。

目前有两种提高 Linux 实时性的技术方案：直接修改内核和增加一个实时内核。除此以外，还有几个其他的改进措施，包括细化 Linux 系统时钟粒度，改善屏蔽中断处理，改善并增加实时调度算法等。目前几种主流的实时 Linux 系统有 RT-Linux、RTAI、Kurt-Linux、Red-Linux、Linux/RK 和 Hard-hat Linux 等[4]。上述的各种实时 Linux，它们针对不同的设计目标，从不同的侧重点解决了通用 Linux 操作系统对实时性支持的问题。对于 Linux 中的封中断的问题， RT-Linux 和 RTAI 所使用的软件模拟中断控制器的方法可以有效地解决这个问题。针对普通 Linux 系统定时粒度过大的问题，可以将 Linux 时钟设置为单次触发的状态，然后利用CPU的时钟计数寄存器提供高达CPU时钟频率的定时精度，Kurt-Linux 采用的就是这种方法。为了解决 Linux 进程在内核态不能被抢占的问题，RED-Linux 在 Linux 内核的很多函数中插入了抢占点原语，使得进程在内核态时，也可以在一定程度上被抢占，另外 Hardhat 也通过修改 spinlock(自旋锁)的宏定义以及中断返回处理代码，实现了一种可抢占的内核。对于 Linux 系统中缺乏实时

调度机制和调度算法的问题，目前有很多新颖的操作系统调度框架和调度算法都有 Linux 实现，比如 RED-Linux 所定义的一个通用的实时调度框架[4]。

调度的实质是一种资源的分配，即如何在系统的各个任务之间合理的分配运行所需要的资源。调度算法决定系统如何进行资源的分配，它是一种服务于系统目标的策略。对于不同的系统，应采取不同的调度算法。常见的通用操作系统所采取的调度算法所追求的设计目标在于优化系统的平均性能，这样的设计目标适合于一般的桌面计算机应用，而不适合解决实时系统中的问题。实时调度面临与原来的通用操作系统调度所不同的考虑，有效的保障系统中每个实时任务的实时性(响应时间、截止期错过率等)是实时调度的一个重要目标。

1.3 论文研究内容和组织结构

在充分研究和分析 Linux 内核调度机制的基础上，本文总结了 Linux 的不足之处以及改善实时性的具体方法，进而在 RTAI 平台之上设计了一种通用调度策略的框架结构，并完成了相关的调度框架结构的设计和实现，本文的主要研究内容和章节结构如下：

第一章是绪论，总体描述了论文的研究背景以及目前嵌入式实时 Linux 的国内外研究现状，介绍了本文的主要研究内容和组织结构。

第二章是实时操作系统的总体介绍，本章系统阐述了实时操作系统的概念和组成，然后对实时操作系统的关键技术进行了论述，详细分析了进程管理、中断处理、进程间同步和互斥以及内存管理等机制。

第三章是 Linux 实时调度机制和 Linux 实时化方案研究，首先叙述了调度的重要性以及评价调度机制主要原则，接着详细分析了 Linux 的内核调度机制，在此基础上总结了 Linux 的实时能力不佳的主要原因，以及目前对 Linux 进行改造的关键技术，分析了几种 Linux 实时化方案的优缺点。

第四章是实时调度算法的研究，首先介绍了实时调度的基本特性、实时调度算法的分类以及三种实时调度策略，接着详细论述了两种比较经典的实时调度算法——RM 算法和 EDF 算法，同时分析了算法在使用过程中遇到的问题及解决方法。

第五章是基于 RTAI 的实时调度框架的设计和改进，是本文的核心内容。首先介绍了 RTAI 关键技术、调度机制和系统结构设计，通过以上分析指出了 RTAI 调度器的优点和局限性。为改变 RTAI 调度的单一性，本文设计了一种具有通用调度策略的框架结构，详述了系统的设计方案及调度框架的改进，接着设计并实现了 RM 和 EDF 调度器，将其添加到系统中，实现了一种通用的调度策略，最后

进行实验，结果验证了改进后的系统具有良好的性能，并扩大了其应用范围。

第六章是结束语，总结了本文的工作以及对未来的展望。

第二章 实时操作系统

2.1 RTOS 概述

实时操作系统是指一个能够在指定时间范围内完成特定的功能或者对外部的异步事件做出响应的操作系统。实时系统与其它普通的系统最大的不同之处就是要满足处理与时间的关系。实时操作系统上的进程执行结果的正确性不仅依赖于计算的逻辑结果，而且还依赖于结果产生的时间[3]。所以实时操作系统必须能够确保其进程对时间的要求，即必须确保在要求的时间内完成制定的任务。在相当一部分嵌入式系统中，系统的实时性在整个系统中起着至关重要的作用。因此，嵌入式操作系统大部分也是实时操作系统。实时操作系统具有如下功能：任务管理、任务间同步和通信、存储器优化管理、实时时钟服务、中断管理服务。实时系统对逻辑和时序的要求非常严格，如果逻辑和时序出现偏差将会引起严重后果。实时系统有两种类型：软实时系统和硬实时系统。软实时系统仅要求事件响应是实时的，并不要求限定某一任务必须在多长时间内完成；而在硬实时系统中，不仅要求任务响应要实时，而且要求在规定的时间内完成事件的处理[5]。通常，大多数实时系统是两者的结合。实时系统的技术关键是如何保证系统的实时性。

2.2 RTOS 的组成

RTOS可根据实际应用环境的要求对内核进行剪裁和重新配置，组成可根据实际的不同应用领域而有所不同，但以下几个重要组成部分是不太变化的：实时内核、网络系统、文件系统和图形接口等[6]，RTOS的体系结构如图2.1所示。

硬件抽象层包含了所有和硬件平台相关的代码，如上下文切换和 I/O 寄存器访问等等。它存在于 RTOS 的最底层，直接访问和控制硬件，对其上层的与机器无关的代码提供访问和控制服务。这样可以简化 RTOS 内核的移植工作，除了设备驱动程序之外，在移植的时候只需要修改 HAL 的代码就可以了。

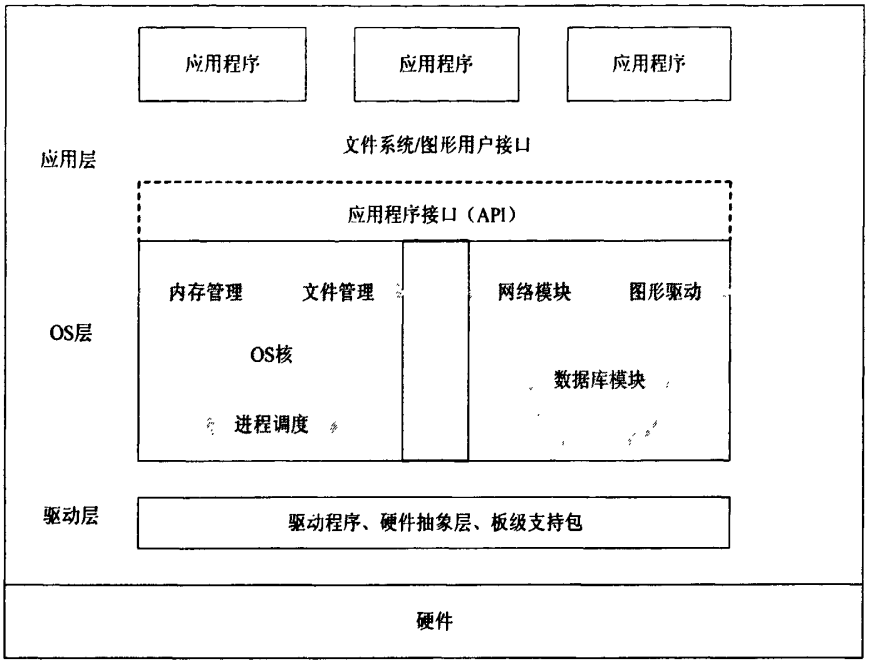

图 2.1 RTOS 的体系结构图

内核是RTOS的基础，也是其最重要的组成部分。RTOS内核是用来为大多数程序乃至网络、文件系统、驱动程序等一系列子系统提供实时支持，使用户程序及上层OS组件对系统设备透明。一般来说，RTOS的内核实现为微内核体系结构。所谓微内核技术是指将必需的功能(如进程管理、任务通信、中断处理、进程调度)放在内核中，而将其它的核心功能和服务(文件系统、存储管理、网络通信、设备管理)等等作为内核之上可配置的部分[5]。在RTOS提供的接口上需要有对用户程序提供的函数接口，专门为用户定制网络、图形、视频等接口。并且提供驱动程序开发界面，方便开发者对不同的设备定制驱动程序。当实际应用需要文件及其管理文件时，RTOS必须支持文件系统。RTOS文件系统主要有文件的存储、检索和更新等功能。一般RTOS文件系统应选择性的支持兼容性，实时性，可裁剪和可配置性以及存储设备多样性等等。

2.3 RTOS 的关键技术

2.3.1 进程管理和调度

实时操作系统的实时性和多任务能力在很大程度上取决于它的任务调度机制。从调度策略上讲，分为优先级调度策略和时间片轮转调度策略；从调度方式上讲，分可抢占、不可抢占、选择可抢占调度方式；从时间片来看，分固定与可变时间片轮转。单纯从基于优先级的抢占式调度方式而台，又存在多种优先级计算方法。

有关进程的数据结构和通用操作系统进程的数据结构肯定有很多的不同，例如加入了优先级继承协议后，进程的数据结构中肯定会有进程本身的优先级和继承的优先级等。

实时系统的中心是短任务调度程序。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必须在它们的最后期限内完成，尽可能多的软实时任务也可以在它们的最后期限之前完成[7]。

大多数通用操作系统都不能直接处理最后期限，它们一般的调度策略都是基于时间片的，对实时任务通常给予很高的优先级，根据POSIX标准，对于具有相同优先级的实时任务，一般操作系统(如Linux)都提供FIFO和round-robin的调度策略，但是仅有这些是远远不够的。

而实时操作系统被设计的尽可能快地对实时任务做出响应，使得当临近最后期限时，一个任务能够迅速的被调度。从这一点看，实时应用程序在许多条件下都要求响应时间在几毫秒到微秒级的范围内。关键应用程序，如军用飞机模拟器通常要求响应时间在 10 到 100 微秒范围内。

在使用简单的循环调度的可抢占式调度程序中，实时任务将被加入到就绪队列中，等待它的下一个时间片。在这种情况下，因为完全没有考虑实时任务的优先级和时间片的循环机制，调度时间通常是实时应用程序难以接受的。在不可抢占的调度程序中，可以使用优先级调度机制，给实时任务更高的优先级，在这种情况下，只要当前的进程阻塞或运行结束，就可以调度这个就绪的实时任务，但是如果在临界资源中，一个比较慢的低优先级任务正在执行，就会导致几秒的延迟。一种比较折衷的办法是把优先级和基于时钟的中断结合起来，抢占点按规则的间隔出现，当出现一个抢占点时，如果有更高优先级的任务正在等待，则当前运行的任务被抢占，这对有些实时应用程序是足够了，但是对一些要求更苛刻的应用程序仍是不够的，这时常常采用一种称作立即抢占的方式，在这种情况下，操作系统几乎是立即响应一个中断，除非系统处于临界代码保护区中，关于实时任务的调度延迟可以降低到100微秒或更少[5]。

另外，在实时操作系统中，不仅需要选择使用一些实时调度算法，为了防止系统过载，还要考虑可调度性分析等。商业实时操作系统都会提供从32-256级的基于固定优先级的实时调度策略。实时调度算法本身就是一门非常复杂的研究方向，也是投入人力最多、研究最为透彻的一个方面。

快速的进程或线程切换也是非常重要的。由于某种原因，使一个任务退出运行时，实时操作系统保存它的运行现场信息、插入相应的队列、并依据定的调度算法重新选择一个任务使之投入运行，这一过程所需的时间称为任务切换时间。

2.3.2 中断管理

中断管理这一部分是操作系统中非常重要和关键的一部分，由于周期性和非周期性实时任务的按时执行都离不开中断，并且大多数实时任务的调度都是由中断引发的，中断管理对于实时系统而言不仅更加重要，而且要求更高。因此，实时系统要求操作系统具备迅速响应外部中断的能力。设计不良的中断管理或磁盘网络操作等可能造成多达几百微秒或十毫秒级的等待，使得整个系统的任务处理延滞，从而导致实时任务得不到及时的响应。

实时操作系统响应中断时，一般都会采用关中断的方式，是不会因为外部中断的到来而中断当前中断响应的。这一过程所需的最大时间是最大中断禁止时间。它和任务切换时间(又称上下文切换时间)一起是评价一个实时操作系统最重要的两个技术指标。

为了最小化关中断的时间，实时操作系统一般采用的方式是将中断处理分为两部分，第一部分的优先级很高，且允许关中断，但是其仅作一些必要的硬件操作，随后设置一些相关的标志位，所以执行的时间非常短，而把流程复杂、耗时的部分放在第二部分执行。第二部分称为中断服务程序，商业系统一般都在系统启动时创建相应的线程在需要时执行之，如果该线程的优先级比较低，而调用该中断的应用程序的优先级又比较高，就有可能发生优先级反转问题，即该中断线程不断的被其他比其优先级高但调用其应用程序的优先级又较低的中断线程不断抢占。在LynxOS中采用了所谓的优先级追踪(priority tracking)技术解决该问题。

时钟中断又是所有中断中最重要的中断，可以认为是操作系统的脉搏，而实时系统最关键的性能就是及时响应用户的请求，没有好的时钟中断管理机制，实时系统就无从谈起。为了提高响应速度，一般都会最小化时钟中断的时间间隔。

一般操作系统的时钟粒度都是10毫秒，这样的时钟粒度对分时系统中的应用而言是必要的而且是合适的，既满足了具体应用程序的需要，又避免了过大的系统开销。但是对于大多数要求响应时间为微秒级的实时应用而言，就过于粗糙了。而简单的增加时钟粒度又会造成系统开销的增加。时钟中断一般分为周期性的时钟中断(Periodic Timer Interrupts)和一次时钟中断(One-Shot Timer Interrupts)[8]，大多数操作系统都是采用周期性时钟中断更新系统时钟计数、减少进程的时间片、检查是否有定时器到期，如果有则执行相应的服务程序。周期性时钟中断的优点是时钟中断处理开销小。如果时钟周期是10毫秒，则很显然，一个周期为5毫秒的实时任务肯定得不到及时的处理。为了提高时钟的粒度，一些实时操作系统(例如QNX)采用了一次时钟中断，将系统的时钟粒度定为微秒级，将下次时钟中断时间设为最早要到期的定时器的到期时间。但是在这种方式下，每次时钟中断时要计算下次时钟中断的时间，同时又要维持系统的周期性，因此会对中断响应有所延

迟。

归纳起来，影响定时器时间不准确性的原因主要有操作系统的时钟粒度、处理定时器服务程序所花的时间等。其中最小不可预测的因素是处理定时器服务程序所花的时间，有些中断服务程序可能会导致长时间的中断延迟。有些操作系统在处理定时器时采用LIFO策略，即优先处理距当前时间最近的到期服务程序，这不符合实时系统的要求。过于粗糙的时钟粒度显然会使得定时器得不到及时的处理。

2.3.3 进程间同步和互斥

进程间的同步和互斥在通用操作系统中也存在，但是没有像在实时系统中这样紧迫。实时操作系统的基本核心都会提供诸如消息、信号量、互斥锁、信号和事件之类的进程间同步和互斥方式。如果由于优先级反转等原因使得高优先级任务不能得到及时响应，那造成的后果可能很严重。所以商业实时操作系统至少都会支持优先级继承协议(PIP)，有的还会提供PCP协议。但是通用操作系统一般不会提供对这两种协议的支持。

2.3.4 内存管理机制

通用的分时操作系统都会支持虚拟内存管理和内存保护功能，但是基于资源和效率的考虑，实时操作系统不一定支持这些功能，即使支持也会让用户选择是否启用这些功能，有些实时系统可能根本就不需要这些功能，下面介绍内存管理的主要机制[9]。

(1) 虚存管理机制。实时操作系统一般都是为嵌入式设备而开发的，这些系统可能仅需要数据采集、信号处理和监视等功能支持虚拟内存管理机制。但是在系统的运行过程中，应用程序对内存需求的变化可能造成内存碎片。虽然采用虚拟地址映射可以让物理上不连续的内存地址在逻辑上连续，以满足需要大地址空间应用程序的需求，但是由于地址映射表也要占用内存，这种方法对资源有限的系统就不太适合了。

(2) 锁内存机制。支持分页机制的操作系统会在系统内存不足的情况下换出部分物理内存页，用户可能需要指定一些实时应用程序的物理页不被换出。POSIX标准指定mlockall()和mlock()函数实现该功能。

(3) 内存保护。一些实时操作系统不提供内存保护功能，将实时应用程序和核心放在一个地址空间运行。这样做的优点是简单，且系统小。但是缺点是任何一点小的改动都需要测试整个系统，而且除了一些非常简单的嵌入式系统，开发费用可能会大幅度增加。所以大多数实时操作系统(QNX, LynxOS) 都提供内存保护。

在VxWorks中，用户可以只选择将部分系统地址空间写保护(例如异常中断向量表，正文段等)，如果任务需要则给其分配私有虚存空间。

(4) 最小内存开销。在实时操作系统的设计过程中，最小内存开销是一个较重要的指标，这是因为实时系统，特别是包括消费类电子产品在内的嵌入式系统，基于降低成本的考虑，其内存的配置一般都不大，而在这有限的空间内不仅要装载实时操作系统，还要装载用户程序。因此，在实时操作系统的设计中，其占用内存大小是一个很重要的指标，这是实时操作系统设计与其它设计的明显区别之一。

2.4 本章小结

本章首先介绍了 RTOS 的概念以及组成，叙述了 RTOS 的内核。然后，介绍了 RTOS 的特点和关键技术，详细分析了进程管理、中断处理、进程间同步和互斥以及内存管理等机制，为下一章对 Linux 系统的分析做了铺垫。

第三章 Linux 调度机制分析及实时方案研究

由于嵌入式实时应用的发展，众多开发人员越来越需要一个优秀的开发环境。早期实时操作系统由于面向特定应用，其扩展特性不好，缺少很多通用操作系统所具有的应用程序以及网络功能。而 Linux 则具有丰富的应用资源，而且其源代码开放特性又非常适合于定制特定的实时系统。因此，很自然的想到使用 Linux 进行实时系统开发，以便于利用它的开发环境、X-windows 以及所有的网络支持。

然而，Linux 本身并不能很好的运行实时应用，为了使 Linux 能够应用于实时环境中，必须对分时的 Linux 进行实时化改造。首先必须对 Linux 系统的内核调度机制有比较清楚的认识，然后才能提出实时化的解决方案并实现，下面对 Linux 的调度机制做了简要的分析。

3.1 调度概述

操作系统内核实现的一项重要功能就是进程调度。进程调度控制和协调进程对 CPU 的竞争，按照一定的调度算法，使某一就绪进程获得 CPU 的控制权，进入运行状态。进程调度系统负责控制进程访问 CPU，是操作系统的关键子系统，其他子系统都依赖于它，因此其性能的好坏关系到整个系统，对操作系统的整体性能有非常重要的影响。一个好的进程调度系统要考虑很多方面的因素：公平、有效、响应时间、周转时间、吞吐量等。Linux kernel 2.4 采用的是时间片轮转和优先级相结合的调度策略[10]，它有以下几个缺陷：调度算法复杂度是 O(n)级；不提供抢占式调度，对实时应用支持较差；在多处理器环境中，只有一个就绪队列，当一个处理器正在对它进行操作时，其他的处理器只能等待，而且负载不能在各处理器之间平衡。目前，最新的 Linux kernel 2.6 对上述几个缺陷都进行了改进，实现了 O(1)的调度算法[11]，支持抢占式调度，增强了多处理器环境下的支持。但是，由于它采用的仍然是基于优先级的任务调度策略，并不能保证实时系统中的低延迟和高度的可预测性，因此，有必要对它再进行改进，以增强其实时性。下面介绍评价进程调度机制主要原则：

(1) 公平：确保每个进程获得合理的 CPU 份额。

(2) 效率：尽量使 CPU 忙碌。

(3) 响应时间：使交互用户尽快获得响应。

(4) 周转时间：使批处理用户等待输出的时间尽可能短。

(5) 吞吐量：使单位时间内处理的作业数尽量多。

(6) 实时性：保证实时任务在截止期限内完成。

这些原则很难同时兼顾，例如，对于分时系统，如果使交互用户尽快获得响应，减少响应时间，就要减少时间片，系统用于上下文切换的时间就增加了，这就降低了单位时间内处理的作业数，即系统的吞吐量减少了。因此，需要针对具体应用环境，重点考虑它们中的一部分原则。

从交互式应用的用户角度，希望周转时间短、响应时间快、截止时间有保证、具有不同的优先权。从充分利用系统的角度，则希望系统吞吐量高、处理机利用率高、各类资源可以得到平衡利用。从实时应用的角度，需要保证实时任务在截止期限内完成。

设计进程调度机制时需要解决的问题[12]：

(1) 调度时机：在什么时间、什么条件下进行调度。

(2) 调度算法：根据什么原则挑选下一个进程进入 CPU 运行。

3.2 Linux 进程调度机制

3.2.1 Linux 进程的调度时机

进程调度的过程，也就是进程状态转换的过程，因此需要分析 Linux 进程的状态信息和状态之间的转换关系。下面是 Linux 在 sched.h 文件中定义的几种进程状态标志[9]：

(1) TASK_RUNNING：运行态标志。表示进程是正在运行的进程(系统的当前进程)或准备运行的进程(在 Running 队列中，等待被安排到系统的 CPU 中运行)。处于该状态的进程实际参与进程调度。

(2) TASK_INTERRUPTIBLE：可中断等待态标志。处于等待队列中的进程，在资源有效时被唤醒，也可被其它进程发出的信号中断睡眠，唤醒后进入就绪状态。

(3) TASK_UNINTERRUPTIBLE：不可中断等待态标志。处于等待队列中的进程，直接等待硬件条件，在资源有效时被唤醒，不可由其它进程通过信号中断唤醒。

(4) TASK_ZOMBIE：僵死态标志。终止的进程，是进程结束运行前的一个过渡状态。虽然此时已经释放了内存、文件等资源，但是在 Task 向量表中仍有一个 task_struct 数据结构项。它不进行任何调度或状态转换，等待父进程将它彻底释放。

(5) TASK_STOPPED：停止态标志。进程被暂停，通过其它进程的信号才能被唤醒。正在调试的进程可以处在该停止状态。

(6) TASK_SWAPPING：换出态标志。进程页面被兑换出内存的进程，这个状态基本上没有用到，只在 sched.c 的 count_active_tasks()函数中判断处于该种状态的进程是否属于 active 的进程。

各个状态标志并不一定表示进程当前的真实状态，例如，当前运行的进程因为等待资源，打算暂时放弃 CPU，它将调用 schedule()来进行进程调度，在调用 schedule()之前，将本进程的状态设置成为 TASK_INTERRUPTIBLE 或 TASK_UNINTERRUPTIBLE，此时这个进程是当前正在运行的进程，而它的状态标志却是等待标志[13]。因此，状态标志实际表示进程申请在未来要进入的状态，各个状态标志之间的转换关系如图 3.1 所示。

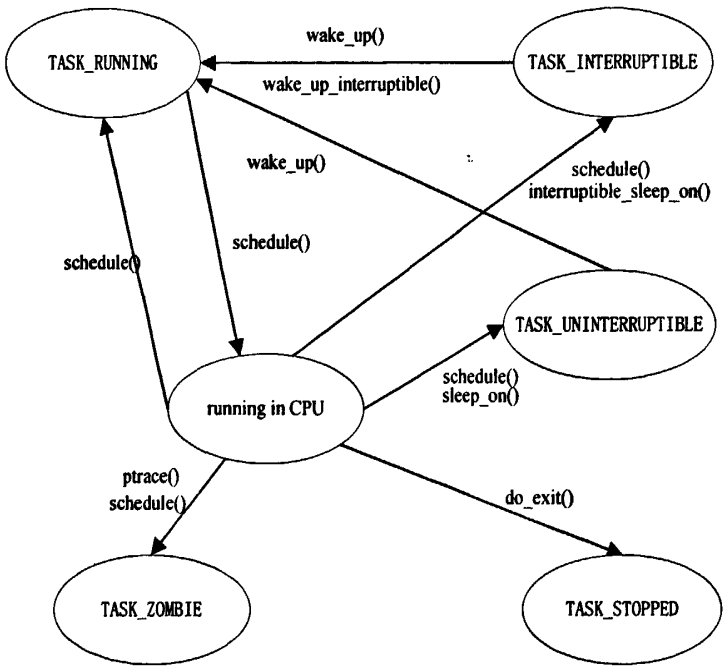

图 3.1 状态标志之间的转换图

在内核空间，一个进程可以通过调用 schedule()启动一次调度，一般在调用 schedule()之前，将本进程的状态设置成为等待标志 TASK_INTERRUPTIBLE 或 TASK_UNINTERRUPTIBLE，暂时放弃运行而进入睡眠，在用户空间，则可以通过系统调用 pause()来实现。在内核空间，还可以通过调用 schedule_timeout()为这种主动的放弃运行加上时间限制，在用户空间则可以通过系统调用 nanosleep()来实现。

entry.S中的代码片段：

```
ENTRY(ret_from_intr)
    GET_CURRENT(%ebx)
    ret_from_exception:
    movl EFLAGS(%esp),%eax        #mix EFLAGS and CS
    movb CS(%esp),%al
    testl $(VM_MASK|3),%eax       #return to VM86 mode or non-supervisor
    jne ret_with_reschedule
```

```
    jmp restore_all
```

寄存器 EAX 的内容有两个来源，其最低的字节来自保存在堆栈中的进入中断前的段寄存器 CS 的内容，最低的两位表示当时的运行级别。从代码中可以看到，转入 ret_with_reschedule 的条件为中断或异常发生前 CPU 的运行级别为 3，即用户状态，所以当 CPU 在内核空间运行时，不会发生强制调度。

从系统空间返回到用户空间只是发生调度的必要条件，而不是充分条件，具体是否发生调度取决于 entry.S 中的一段代码。

```
ENTRY(ret_from_sys_call)
    cli                              # need_esresched and signals atomic test
    cmpt $0,need_resched(%ebx)
    jne reschedule
    cmpl $O,sigpending(%ebx)
    jne signal_return
restore all:
    RESTORE_ALL
.....
Reschedule:
    call SYMBOL_NAME(schedule)
    jmp ret_from_sys_call
#test
```

分析这段代码可知，只有在当前进程的 task_struct 结构中的 need_resched 不是 0 时，才能够调用 schedule()，而 task_struct 结构中的 need_resched 是在如下的三种情况下程序运行的结果。

(1) 发生时钟中断时，程序 do_timer_interrupt()调用程序 do_timer()，接着，程序 do_timer()再调用程序 update_process_times()，在 update_processes_times()中将 need_resched 设置为 1。

(2) 在内核中唤醒一个睡眠的进程时，调用程序 wake_up_process()，然后，wake_up_process()再调用 reschedule_idle()，在 reschedule_idle()中将 need_resched 设置为 1。

(3) 系统调用 set_scheduler()和 sched_yield()在执行的过程中将 need_resched 设置为 1。

由上面的分析可以知道，Linux 进程的调度时机发生在以下情况[13]：

(1) 当前运行的进程自愿放弃运行。

(2) 系统调用返回前的时刻，以及从中断或者异常处理返回到用户空间之前的时刻。

(3) 采用轮转法(round robin)调度的进程时间片到时(10ms 的整数倍)，由时钟中断引起新一轮调度。

3.2.2 Linux 进程的调度策略

Linux 使用基于优先级的调度算法[14]，在系统当前所有可以运行的进程中选择优先权值最高的进程进入 CPU 运行。Linux 中把用户进程分为非实时进程和实时进程两类，非实时进程有两种优先级，静态优先级和动态优先级，实时进程又增加了实时优先级。

内核根据 task_struct 结构中和进程调度相关的域(如表 3.1 所示)进行进程的调度[13]。

表 3.1 task_struct 结构中与进程调度相关的域

域	描述
policy(策略)	系统对该进程实施的调度策略。包括SCHED_FIFO，SCHED_RR，SCHED_OTHER
priority(优先级)	系统为进程设置的优先级。优先级实际是从进程开始运行的时刻为起点，允许进程运行的时间值(以jifies为单位)
rt_priority(实时优先级)	系统为实时进程设置的相对优先级
counter(计数器)	进程运行的时间值(以jifies为单位)。开始运行时设置为priority，每次时钟中断该值减1

Linux 进程的调度策略 policy 可以调用 sys_setscheduler()改变，它的取值有 3 个[10]：

(1) SCHED_FIFO：实时进程先进先出调度策略，当前实时进程会一直运行，直到运行完才释放 CPU，或者是 CPU 被另一个具有更高 rt_priority 的实时进程抢占。

(2) SCHED_RR：实时进程轮转调度策略，除了时间片有些不同之外，它和 SCHED_FIFO 类似。当 SCHED_RR 进程的时间片用完后，就跳转到 SCHED_FIFO 和 SCHED_RR 列表的最后。

(3) SCHED_OTHER：非实时进程基于优先级的轮转调度策略。

schedule()对可运行进程队列调度，从而决定了哪个进程进入 CPU 运行。对于当前进程的调度策略是 SCHED_RR 进程先做如下的调度：

```
if (unlikely(prev->policy = SCHED_ RR))
    if (!prev->counter){
        prev->counter= NICE_TO_TICKS(prev->nice);
        move_last_runqueue(prev);
    }
```

prev->counter 是当前进程的运行时间配额，在每次时钟中断时，由 update_process_times()将 prev->counter 的数值递减。因此，随着运行时间的积累它最终总会递减到 0。对于调度策略是 SCHED_RR 的进程，当它的时间配额降到 0

时，内核就要把它从可执行进程队列 runqueue 中当前的位置移到队列的末尾，同时恢复其最初的时间配额。对于具有相同优先级的进程，在调度的时候，排在前面的进程优先，所以这可以让队列中具有相同优先级的其它进程有进入 CPU 运行的机会。NICE_TO_TICKS 将进程的优先级别换算成时间配额。

接着按照当前进程的状态 prev->state 做如下的处理：

```
switch(prev->state){
    case TASK_INTERRUPTIBLE:
        if(signal_pending(prev)){
            prev->state = TASK_RUNNING;
            break;
        }
    Default:
        del_from_runqueue(prev);
    case TASK_RUNNING:;
        prev->need_resched=0;
}
```

当前进程虽然是正在运行的进程，但是它的状态标志 state 却不一定是 TASK_RUNNING，因为当前进程在主动放弃 CPU 时，会将自己的状态标志 state 改为其它的值。所以 schedule()对于状态标志 state 是 TASK_INTERRUPTIBLE 和 TASK_ZOMBE 的进程， 就通过 del_from_runqueue()将它从可执行队列中删除。对于 TASK_INTERRUPTIBLE 的进程，在有信号等待处理时要将其改成 TASK_RUNNING，在没有信号等待处理时，也通过 del_from_runqueue()将它从可执行队列中删除。对于当前进程的状态标志 state 是 TASK_RUNNING，继续进行。因为系统已经在调度程序了，所以将 prev->need_resched 恢复成 0[10]。

然后，schedule()对可执行进程队列 runqueue 进行操作，具体决定了哪个进程进入 CPU 运行：

```
repeat_schedule:
/*
    *Defaultpr ocessto s elect..
*/
next=idle_task(this_cpu);
c=-1000;
list_for_each(tmp,&runqueue_head){
    p=list_entry(tmp,struct task_struct,run_list);
    if(can_schedule(p,this_cpu)){
        int weight=goodness(p,this_cpu,prev->active_mm);
        if(weight>c)
            c=weight, next=p;
    }
}
```

可以看到 schedule()遍历可执行队列 runqueue 中的每个进程，选择权值 weight 最大的进程，而 weight 则是通过函数 goodness()计算出的。

函数goodness()的代码如下：

```
Static inline int goodness(struct task_struct * p,int this_cpu,struct mm_struct this_mm)
{
int weight;
```

```
/*
    *select the current process after every other
    *runnable process,but before the idle thread.
    *Also,don't trigger a counter recalculation.
*/
weight=1;
if (P->policy&SCHED_YIELD)
    goto out;
    /*
        *Non-RT process_normal case first.
    */
if（p->policy—SCHED_OTHER){
    /*
        *Give the process a first-approximation goodness value
        *according to the number of clock-ticks it has left.
        *Don't do any other calculations if the time slice is
        *over..
    */
weight=p->counter;
if (!weight)
    goto out;
#ifdef CONFIG_SMP
    /*Give a largish advantage to the same processor... */
    /*( this is equivalent to penalizing other processors)*/
if(p->processor==this_cpu)
    weight+=PROC_CHANGE_PENALTY;
#endif
/*..and a slight advantage to the current MM*/
if(p->mm==this_mm||!p->mm)
    weight+=1;
weight+=20-p->nice;
goto out;
}
    /*
        *Realtime process,select the first one on the
        *runqueue (taking priorities within processes
        *into account.
    */
weight=1000+p->rt_priority;
out:
    return weight;
}
```

分析这段代码可知，如果一个进程通过系统调用 sched_yield()将它的 policy 设置成 SCHED_YIELD，则它的权值为-1。

对于调度政策为 SCHED_OTHER 的进程，其权值主要取决于时间配额 p->counter 和进程的优先级 nice。如果是个内核线程，或者其用户空间与当前进程相同，因而无需切换用户空间，则将权值加 1。

对于实时进程，即调度策略为 SCHED_FIFO 或 SCHED_RR 的进程，它比普通进程增加了实时优先级 rt_priority，其权值为 1000 加上实时优先级 rt_priority。可见实时进程的权值至少是 1000，所以当有实时进程就绪时非实时进程是没有机会运行的[13]。

3.3 Linux 操作系统实时性能不足的原因及改进方法

3.3.1 Linux 实时性能不佳的主要原因

Linux 操作系统作为通用操作系统，它的设计目标是取得最优平均性能，因此有很多方面无法满足实时系统的要求。通过对 Linux 操作系统的研究，可以看出 Linux 操作系统在实时环境中的问题主要有以下几个方面：

(1) 内核不可抢占的问题：当一低优先级的进程由于调用系统而进入核心状态后，除非当前进程需要等待资源释放而挂起，否则后来的高优先级的进程只能等待当前进程完成系统调用，而系统调用的完成时间有很大的不可预测性，从而发生优先级反转的问题，导致实时应用响应时间的不可预测性[15]。

(2) 内存换出的问题：Linux 采用了虚拟内存管理技术，进程运行所需的内存常常会被换入换出，如果一个实时任务换出内存，当再次调度它运行时，必须首先经历一个时间不确定的换入过程，从而导致实时任务响应时间的不确定性。实时应用有时需要把关键进程锁在内存中，不被换出，而标准的 Linux 无法满足这种要求。

(3) 关中断问题：在系统调用中，为了保护临界区资源，Linux 会长时间关掉中断，这样会加大中断延迟时间，阻塞高优先级的中断立即被处理。

(4) 时钟精度的问题：操作系统必须对时间精度和时钟中断处理的时间开销进行折中考虑，时间精度越高，意味着时钟中断越频繁，而花在中断处理上的时间越多，Linux 通过对硬件时钟编程产生周期为 100Hz 的时钟中断，因此任务调度的时间精度最高能达到 10ms，无法满足一些对时间精度要求苛刻的实时应用[16]。

(5) 实时任务没有质量保证：虽然 Linux 内核具有一定的实时性特征，但是还不能达到硬实时系统的要求。Linux 允许一个进程被定义为实时进程，但对于这种实时进程，调度器只是简单地给它赋予一个比非实时进程优先级高的优先级，而没有保证对实时进程的响应时间。一些低优先级的进程还是会阻塞实时进程的执行。

3.3.2 Linux 实时性能改进的主要方法

由于 Linux 在实时应用方面存在的问题，使其还无法满足工业上的实时性要求，因此有必要对其进行实时性改造，增强其硬实时性。常用的增强 Linux 实时性的方法，主要从以下几个方面进行：增加 Linux 内核可抢占性、细化 Linux 系统时钟粒度、改善屏蔽中断处理、改善并增加实时调度算法等等。下面，简单讨论一下这些增强 Linux 实时性的方法。

(1) 增强 Linux 内核抢占性。当进程运行在核心态下时，有 2 种实现 Linux 内核抢占的思想：一是完全剥夺抢占方式，另外一个是插入抢占点的方式。完全剥夺方式核心思想是利用了 Linux 支持 SMP 的实现——允许多个进程运行在核心态，通过信号量锁或自旋锁 spin-locks 对内核数据结构存取访问进行保护的原理，实现内核的可抢占调度。插入抢占点方法是在 Linux 内核插入一些抢占点，当内核执行到抢占点，就检查是否有更高优先级的实时进程正在等待，如果有，当前执行进程将被挂起，转去执行更高优先级的实时进程；如果没有更高优先级的实时进程等待，则当前进程将继续执行[16]。

(2) 细化 Linux 时钟粒度。标准 Linux 系统的时钟粒度为 10ms，不能满足系统实时调度的需求，因此需要细化其时钟粒度，具体可以通过两种方式加以实现：一是通过将系统的实时时钟芯片置于单次触发模式，提供十几个微秒级的调度粒度，从而细化时钟粒度；二是通过修改 Linux 内核中宏的定义来细化时钟粒度，如标准 Linux 中 Hz 为 100，如果将其改为 100000，则时钟粒度可以达到 10μs，这种方式虽然会增加许多的系统开销，但在强周期性环境下，对定时器的设置只需进行一次初始化，这样就在一定程度上保证了处理效率[17]。

(3) 屏蔽中断的处理。由于 Linux 在中断处理及虚存管理的缺页处理过程中允许屏蔽中断，这就使得 Linux 系统实时调度的粒度比较大，为此，通常用中断抽象层方法加以解决，其基本思想是尽量不对 Linux 内核源代码进行修改，增加的中断抽象层可以用来截获硬件发出的中断请求，从而完全控制硬件中断，并产生模拟中断信号，发送给 Linux 内核，在系统中有独立的调度器，Linux 内核作为优先级最低的任务被调度器调度，从而增强 Linux 系统实时调度功能。这里的中断抽象层既可以是软件实现的仿真层，也可以是硬件抽象层[18]。

(4) 改善 Linux 内核实时调度算法及策略。目前主要的实时调度算法在下一章给出了具体的介绍。

3.4 Linux 实时化方案研究

3.4.1 单内核结构的实时 Linux

经过 POSIX 标准化以后，Linux 本身的实时性能仍然不足，这也限制了它的应用。为此，人们在对 Linux 内核进行了大量的修改，开发出很多实时操作系统。由于 Linux 本身是单内核结构的，这些通过修改 Linux 内核而得到的实时操作系统可以看成是单内核结构的，下面介绍单内核结构的实时 Linux 系统——RED-Linux。

RED-Linux(Real-time and Embedded Linux)是在加州大学艾尔文分校开发的一个软实时操作系统。RED-Linux 也是基于 Linux 操作系统的，但是，为了避免将

Linux 内核改造为完全可抢先及可重入所带来的巨大工作量，开发者只是在 Linux 内核代码中加入很多可抢先点。这样通过将代码分割为很多小片，来减少内核抢先所带来的延迟。这种途径并不提供像后面将介绍的 RTAI， RT-Linux 那样的定时精度，但是很明显，在该系统中，可以直接访问 Linux 内核服务，这为实时与非实时进程间通信提供一个统一的接口。由上可见，RED-Linux 可以提高 Linux 内核的实时能力，同时保留所有现存的服务。在内核部分可抢先化之后，RED-Linux 又为 Linux 添加了三个特性：一个源自于 RT-Linux 的高精度定时器，一个软件中断模拟，以及一个通用实时调度器框架。这个框架可以支持三种最常见的调度模式：优先级驱动、时间驱动、以及共享驱动[4] [19]。

RED-Linux 的设计目标就是提供一个可以支持各种调度算法的通用的调度框架，该系统给每个任务增加了 Priority(作业的优先级)、 Start-Time(作业的开始时间)、 Finish-Time(作业的结束时间)、Budget(作业在运行期间所要使用的资源的多少)这几项属性，并将它们作为进程调度的依据。通过调整这些属性的取值及调度程序按照什么样的优先顺序来使用这些属性值，几乎可以实现所有的调度算法。这样的话，可以将三种不同的调度算法统一地结合到了一起，RED-Linux 调度程序的框架结构如图 3.2 所示。

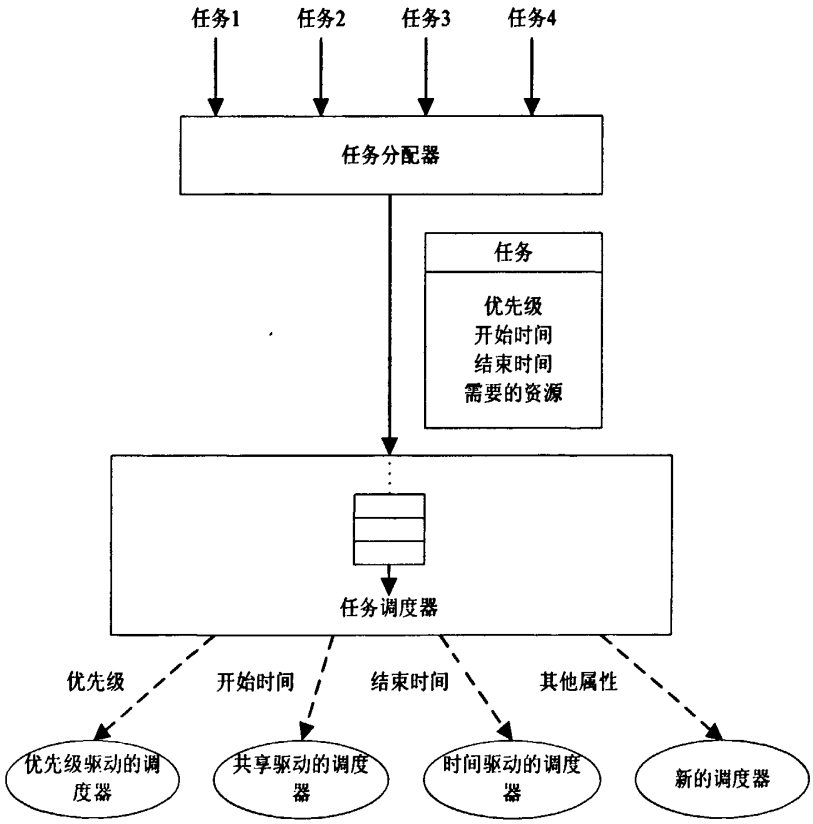

图 3.2 RED-Linux 调度框架图

3.4.2 双内核结构的实时 Linux

如果需要将 Linux 内核改变成真正的可抢先实时内核，以减小中断处理潜伏期，那么就需要从根本上对 Linux 内核代码进行重写，这所带来的工作量相当于写一个新的操作系统。以上所介绍的基于单内核结构的实时操作系统并没有这样做。它们只是通过修改了部分 Linux 内核来减少工作量，同时提供实时性能。然而，由于 Linux 内核本身的非抢占性，上述修改往往只能提供一定的软实时能力。基于双内核结构的实时操作系统。则采用了一个更简单和更有效率的解决方案。在这种系统中，实时部分和非实时部分被明确的分开。这种去耦合的机制使得实时任务变得更加简单、快速，并具有很好的预期性。同时，为了支持非实时应用，Linux 内核被作为最低优先级的实时任务而运行[20]。这样的操作系统目前有 RT-Linux 和 RTAI。

RT-Linux 是由美国新墨西哥州的 New Mexico Institute of Technology 计算机系研制开发而成，是一个基于 Linux 的硬实时系统。为了增加一个实时内核，达到实时化 Linux 的目的，RT-Linux 采用了如下方式[20]：

(1) 对 Linux 核心进行改动，将其与中断控制器隔离，不再允许它任意关中断。核心中的所有中断操作指令都被替换为相应的宏，可以简单地理解为，这时的开中断、关中断指令实际上仅仅变更一个中断状态标志的值，并不真正改变中断状态。此时，中断控制器由实时核心控制，所有的中断首先被实时核心所截获，并进行相关的中断处理，然后才把中断传给原有 Linux 核心。这样，原有 Linux 核心的一切活动都无法导致中断被关闭，也就无从影响实时核心的任务调度，从而保证了核心的硬实时性。同时，容易看出，这种方法依然维护了 Linux 核心中数据结构的完整性。

(2) 改变了时钟中断机制。与 KURT 一样，RT-Linux 也需要用粒度更细的时钟。事实上，RT-Linux 采用的方法与 KURT 是一样的，将时钟中断设置为单次触发模式。

(3) 提供实时调度。人们对实时任务调度问题进行了长期而深入的研究。实际上，这是实时操作系统领域内投入最多、研究最为透彻的一个方面。目前已经提出了形形色色的调度算法。这些算法适用于不同类型的实时应用场合，各有千秋。但是，要在一个具体实时操作系统内部实现一个通用的调度模块也不是不可能的，只是效率会低些。RT-Linux 在这方面采用了灵活的做法。使用者可以根据具体情况编制自己的调度模块，然后很容易将编好的模块加入到系统中。这样，使用者可以根据自己的不同需求，采用适宜的调度算法。

(4) 实时进程与非实时进程间的通信。如前所述，RT-Linux 的全部设计思想基于实时应用的划分。这里，一个实时应用被划分成一个运行于实时核心之上的实

时进程以及运行于 Linux 核心之上的分时进程。于是，实时进程与非实时进程间的通信就成了一个必须解决的问题。RT-Linux 通过建立命名管道和共享内存来解决这个问题，在实时进程与非实时进程之间建立了数据传输机制。

出于以上的考虑，设计者将 RT-Linux 设计成一个小的实时管理器，它基于 Linux，但是与 Linux 内核独立，从而与 Linux 内核一起构成一个双内核系统。它将 Linux 内核作为优先级最低的任务运行。在 RT-Linux 中，Linux 内核与实时任务一起由专门的实时调度器进行管理。实时任务可以直接访问硬件资源，这大大提高了对外部事件的处理速度，RT-Linux 的基本结构如图 3.3 所示。

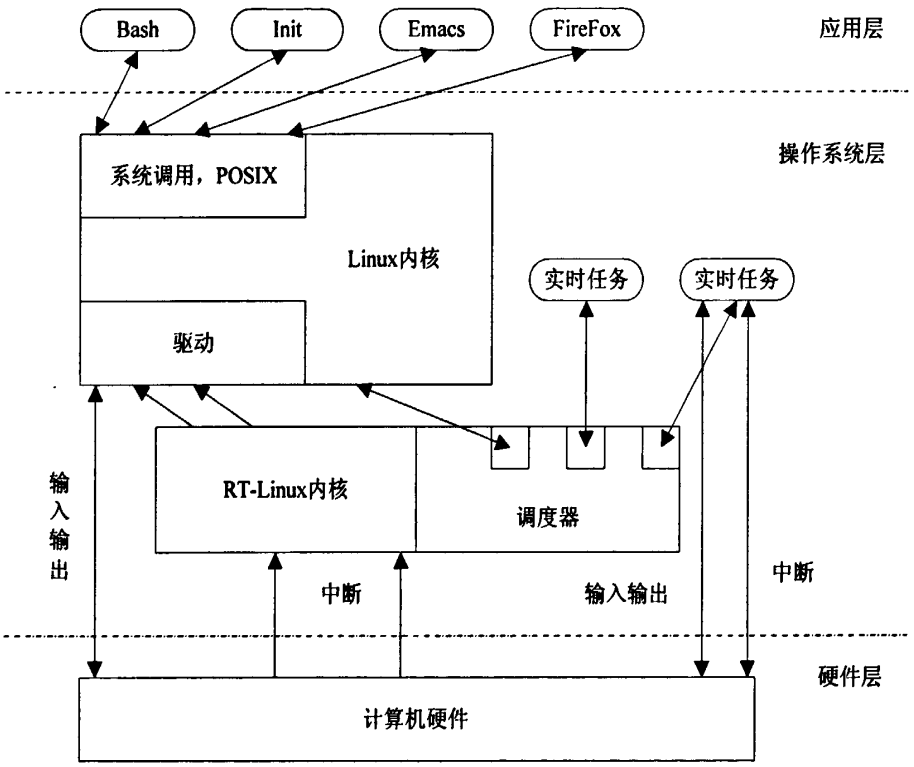

图 3.3 RT-Linux 结构框图

3.5 本章小结

本章首先介绍了调度对操作系统的重要作用，以及评价调度机制的主要原则。接着对 Linux 的进程调度时机和算法进行了详细的分析，其中 Linux 的进程调度算法是本文关注的重点。通过以上分析，得出了 Linux 实时性能不佳的主要原因，并总结了目前改造 Linux 的主要技术。然后着重研究了 Linux 主流的实时化方案，其中的 RED-Linux 和 RTAI 是本文的研究的重点。

第四章 实时调度算法研究

对实时调度的研究开始于二十世纪七十年代中期，在近长十年的时间里，该领域的研究空前活跃，获得了飞速的发展。从早期的研究一直到现在，很多的实时调度算法都是针对单机系统提出的。我们可以把对单机实时系统的调度算法研究划分成静态调度和动态调度两大类别，共四种模式。首先，在静态调度中，调度算法要事先知道实时任务的各种特性，如截止期、运行时间、就绪时间和时序约束。RM(Rate Monotonic)及其扩展算法是这一方面的典型代表。这些算法共同构成了第一种模式。其次，动态调度又可进一步被分为资源充足环境下的调度和资源不足环境下的调度。在资源充足环境中，尽管实时任务是动态到达的，但系统中的资源仍能够保证在任何时刻所有的任务都是可调度的。在一定的条件下，EDF(Earliest Deadline First)算法是资源充足环境下的最优动态调度算法。EDF 算法构成了实时调度的第二种模式。但是，在资源不足的环境下， EDF 算法的性能将会大幅度下降并出现过载现象。另外，其它的一些算法，如 RED 算法也可用于资源不足环境下的调度。这些基于计划的调度算法构成了实时调度的第三种模式。近几年，针对实时应用环境大多不可预测，且任务的工作负载和时间约束常为不明确的这一特点，又提出了一些闭环式的实时调度算法。这些算法形成了实时调度的第四种模式[22]。下面首先介绍一下实时调度的基本知识。

4.1 实时调度的基本知识

4.1.1 任务及其特性

任务是指完成某一特定功能的软件实体，它是实时调度中的基本单位。从调度的角度来看，一个实时任务具有如下基本特性：

(1) 任务到达时间
(2) 任务就绪时间
(3) 任务运行时间
(4) 任务截止时间
(5) 任务到达频率

根据任务到达频率的不同，任务可以被分为周期性任务和非周期性任务。对于周期性任务，其相邻两次到达时间之间的间隔是一个固定的常数(即周期)。这样，一旦获知了任务的首次到达时间，它今后的每次到达时间用首次到达时间加上周

期的若干倍数即可得到。非周期性任务的相邻两次到达时间的间隔是任意的，没有任何限制。同时，这类任务的到达时间事先是不可预测的，且其特性在任务到达前是不知道的。

此外，任务还具有其它一些与调度相关的特性：

(1) 可开始时间

该时间即为任务运行所需资源均为空闲的时间，它是由调度以及任务间的约束关系决定的。

(2) 等待时间

这是任务为了取得运行权而需要等待的时间。它是由于其它更高级别任务的抢占或由于该任务要访问一些共享资源而被阻塞所造成的，其大小通常是由调度器的实现方法决定的。

(3) 转向时间

这是指任务从就绪到完成的时间。如果忽略开销，则这一时间为任务运行时间与等待时间之和。

同时，根据任务重要性的不同，还可以把实时任务分为硬实时任务和软实时任务两种。硬实时任务具有硬时间约束，其截止期的错过将会导致严重的后果。软实时任务则具有软时间约束，其截止期的错过在某种程度下是可以被接受的，不会引起非常严重的后果。

4.1.2 任务间的相关性

任务间的相关性有以下几种[23]：

(1) 顺序相关性：这是指某些任务之间必须按照一定的先后顺序开始运行。它是由与应用相关的处理顺序、任务间交换数据的需要或任务间通讯而导致的。如果任务间不存在顺序相关性，则称这些任务是独立的。

(2) 资源相关性：它包括任务互斥地使用主动资源及任务共享或互斥地使用被动资源。

(3) 优先级顺序：高优先级的任务可以抢占低优先级任务的运行。此外，当若干个任务同时发出运行请求时，具有最高优先级的任务将首先占有处理器。

(4) 与策略相关的约束：这主要是指任务的可抢占性，而且是由应用的性质所决定的。对于可抢占任务，如果必要，它在任何时刻都可被其它任务中断；对于非抢占任务，一旦它开始运行，则在它完成之前都不能被中断。

(5) 策略导致的约束：这是由应用的额外要求所引起的。如某些任务必须要在某个处理器上运行或者某些任务必须要在某个特定时间运行等。

4.1.3 实时调度算法的分类

实时调度算法的分类可有多种方法。常用分类方法如表4.1所示[24]。

表4.1 实时调度的分类方法

分类依据	分类
根据所解决的问题性质	静态调度和动态调度
根据实时任务是否可被抢占	抢占式调度和非抢占式调
根据实时系统运行环境	单处理器实时调度和多处理器实时调度
根据调度器体系结构	集中式调度和分布式调度

(1) 静态调度和动态调度

若调度算法是在编译的时候就做出决定从就绪任务队列中选择哪个任务来运行，则这样的调度是静态的。这类调度算法假设系统中实时任务的特性是事先知道的。它脱机地进行可调度性分析，并产生一个调度表。调度器在运行时将依照这张表所提供的信息选择任务来运行。静态调度适合于问题需求比较确定的情况，如工业过程控制。静态调度算法的优点是运行开销小，可预测性强，一旦找到了一个调度解决方案，便能够保证所有任务的截止期都可以被满足。但是，由于静态调度算法一旦做出调度决定后在运行期间就不能再改变了，所以它的灵活性较差，不适合不可预测环境下的调度。

如果调度器是在运行期间才决定选择哪个就绪任务来运行，则这类调度被称为动态调度。它只考虑目前就绪任务集中任务的特性，以此来决定当前的调度序列，对未来将要到达的任务的特性却一无所知。动态调度算法需要对变化的环境做出反应，因此，这类调度算法比较灵活，适合于任务不断生成，且在任务生成前其特性并不清楚的动态实时系统。但是，动态调度算法的可预测性差且运行开销较大。

(2) 抢占式调度和非抢占式调度

在抢占式调度中，目前正在运行的任务可以被别的更紧迫和更重要的任务中断。同时，被抢占的任务在未来可以恢复运行，且不会影响到任务的整体时限约束。这种抢占式的调度常见于工业制造中。它的优点是比较灵活，对资源的利用率高；但由于经常出现的上下文切换使得其开销较大。

如果在调度中不允许正在运行的任务被别的任务中断，任务一旦占有了处理器便会一直运行直至完成，这样的调度则是非抢占式的，它比较适合于任务运行时间都比较短的系统。其优点是省去了进行上下文切换的开销；同时，具有更好的可预测性，更易于测试，在保证对共享资源的互斥访问上也有较强的继承能力。非抢占式调度没有抢占式调度那样灵活，对资源的利用率也相对较低。

(3) 单处理器实时调度和多处理器实时调度

如果实时系统运行在一个单独的处理器上，则为实时单处理器系统。在这种系统中对实时任务进行调度时，仅需要考虑任务在这个处理器上的运行情况。如果实时系统运行在一个多处理器的环境下，则为实时多处理器系统，此时在进行实时调度时，不仅要考虑任务在单个处理器上的运行情况，还要考虑任务被分配到哪一个处理器上运行，即任务分配问题[25]。

(4) 集中式调度和分布式调度

这种分类是针对实时多处理器系统的调度而进行的。在集中式调度中，系统中有一个专门的处理器作为调度器。所有的任务都到达这个中心调度器，由它来做统一的调度决定，然后这些任务再从这里被分配到其它处理器上去运行。在分布式调度中，每个节点上都有一个调度器，这些调度器在系统中的地位相同，它们可以各自做出调度决策。若任务在某个节点上无法被调度，则可以将该任务迁移到其它可以保证其截止期的节点上去运行。

4.1.4 实时调度策略

在实时操作系统中，应用进程对时间限制有着明确的需求，进程调度策略是实时系统内核的关键部分，如何进行合理的任务调度，使得各个任务能够在其规定的期限之内完成，是实时操作系统的一个重要研究领域。对于一个实时操作系统来说，它的一个重要任务就是利用某些实时调度算法来协调系统资源的使用并满足系统的时间限制。其中调度算法必须实现以下目标：

(1) 保证硬实时任务总是在其时限之前完成。

(2) 对于硬实时任务，系统将达到一个高的可调度利用率，所谓可调度利用率就是指在满足硬实时限制的前提下，最大的系统资源利用率。

(3) 为软实时任务提供最快平均响应时间。

(4) 在瞬间超载时，确保调度的稳定性。调度稳定性就是指在系统超负荷时，虽然某些任务的期限将会被错过，但是必须保证关键任务的期限要求。不同的实时应用系统的用户可能希望用不同的策略来调度，因此，对于实时操作系统来讲最好是把调度和调度机制分开。实时操作系统提供实现某些调度策略的算法，并把调度策略参数化，让用户根据不同的需要选择所需的调度策略。

常用的实时调度策略有：时间驱动调度策略、共享驱动调度策略和优先级调度策略[26]。

(1) 时间驱动调度策略

时间驱动的调度算法中的典型算法是基于时间的静态表调度算法。基于时间的静态表调度算法首先对给定任务的时间特性进行分析，然后产生固定的调度表。任务执行时，根据这个调度表来决定调度任务的执行顺序、执行时间、开始时刻

和完成时刻，并且进程执行顺序都是固定的，不允许抢占。表的覆盖长度应至少为所有任务周期的最小公倍数。如果任务之间不是独立的，有前趋、互斥、同步、通信等约束条件时，则调度是NP难问题，需要采用启发式算法，通常是采用分支和有界搜索算法寻找可行的调度。静态调度表的优点是可预测性高，运行时的调度开销小，使用在简单但安全性极高的系统。缺点是在线调度不灵活，所有任务执行顺序必须严格地按照事先制订的调度表来进行。当任务数或任务特性发生改变时，必须修改整个调度表，因此很难处理突发情形。

(2) 共享驱动调度策略

共享驱动的调度算法按照GPS(General Processor Sharing)模型来实现的。运行队列中的所有任务按一定的比例共享系统资源。著名的加权公平队列调度WFQ(Weighted Fair Queueing)就是共享驱动的调度算法。WFQ按照最小虚拟完成时间决定服务的顺序。实现共享驱动的调度算法很复杂，系统总要不停地更新每个任务的虚拟完成时间，这增加了系统调度的计算量，降低了系统的效率。由于共享驱动的调度算法没有定义优先级，所有任务按照它们的比例都是平等的，当系统发生过载时，所有任务都会按照它们的共享比例发生延时。

(3) 优先级驱动策略

优先级驱动的调度算法是通过给任务分配优先级来实现的。在任务调度的时刻，选择优先级最高的任务进入 CPU 运行。优先级分为静态优先级和动态优先级。著名的单调比率算法(RM)是基于静态优先级的调度算法，最早截止优先算法(EDF)是基于动态优先级的调度算法。单调比率算法(RM)：此算法给系统中每个任务设置一个静态的优先级，这个优先级的设定是在计算任务的周期性和任务需要满足死线时间的长短这两个因素的基础上来完成的。周期越短，时间越紧迫，优先级越高。调度程序总是调度优先级最高的就绪进程，必要时可以剥夺当前进程。最早截止优先算法(EDF)：此算法根据任务满足死线的紧迫性来修改任务的优先级，从而保证最紧迫的任务能够及时地完成。当系统的负载相对较低时，这种算法非常有效。但是，当系统中发生了瞬时过载时，系统的行为是不可预测的，从而使 CPU 时间大量花费在调度上，在这时系统的性能急剧下降。由于优先级驱动的调度算法总是让高优先级的任务运行，可能导致错误的高优先级任务过多地占用 CPU 时间，减少了低优先级任务的运行时间，所以系统需要增加判断任务是否允许进入运行队列的控制机制。另外，优先级驱动的调度算法根据时间因素来确定任务的优先级，例如，单调比率算法 (RM)根据周期的长短，最早截止优先算法(EDF)根据截止时间的早晚。所以它很难调度具有长的周期或者较晚的截止时间，但是却需要快速响应的任务[4]。

4.2 RM 算法

基于优先级的调度策略是非常重要的一种调度策略。按照分配优先级的方式可将其分为动态优先级算法和固定优先级算法。固定优先级算法为每个任务中指定一个固定不变的优先级。主要算法有RM(Rate-Monotonic)算法和DM(Deadline Monotonic)算法。RM算法指定周期最短的任务优先级最高，周期越短，任务的到达频率越高。而DM算法指定相对截至期限(deadline)短的任务优先级高。其中RM算法在1973年就提出了，但是在最近得到普及，并广泛的应用于工业系统中，以下将主要介绍与RM算法有关的一些理论和实际应用的研究。

4.2.1 算法概述

RM算法事先为每个实时任务分配一个与事件频率成正比的优先级。运行时，调度程序总是调度优先级最高的就绪任务，必要时抢占当前正在运行的任务。实现时，就绪队列中的所有任务，按优先级排队，优先级最高的任务排在队首。当处于运行态的任务由于某种原因而挂起时，只要把就绪队列的首元素从就绪队列中取下，使运行任务指针指向该元素即可。如果是处于其他状态的任务变为就绪状态而挂于就绪队列时，则必须对运行任务和就绪队列首元素的任务进行比较，优先级高的任务先运行[27]。

RM算法是最理想的基于单处理器的静态优先级的可抢占式调度算法，也就是说，当任意一种静态优先级算法可以产生可行的调度时，RM算法也可以。

RM算法基于以下假设：

(1) 没有任务具有不可优先抢占段，而且优先抢占的耗费是可以忽略的。

(2) 只有数据处理的需求是重要的，内存、1/0和其他资源需求是可忽略的。

(3) 所有任务都是独立的，同时不存在优先约束。

假设1表明：可以在任何时候抢占任何任务并且在不付出任何代价的情况下恢复该任务，所以被强占的任务次数并不会改变处理器总的工作量。

假设2表明：对于任务的可行性，只需确保系统有足够的处理能力可以执行任务并且使任务满足其最终期限，而内存或者其他的约束条件并不能使问题复杂化.

假设3表明：任意任务释放的次数并不依赖于其他任务的完成次数。

为了方便讨论，除了上述的三个基本假设之外，一般都辅以下面的两个假设：

(4) 任务集合中的所有任务都是周期性的。

(5) 任务的相对时限等于它的周期。

假设 4：为讨论方便所做出的假设，对非周期性任务的调度问题有补充讨论。

假设5：用以简化对RM算法的分析，它确保在任何时候，任何有效的任务最

多只有一次迭代。

4.2.2 可调度性分析

当存在于一个任务集里的任意一个任务在任何时刻系统都可以满足它的时间要求，那么这个任务集可以被称为可调度的任务集[1]。

RM算法分配给每一个任务一个固定的优先级来最大化任务集的可调度性。静态优先级调度算法的一个最大的局限就是它不可能总是能够使得CPU完全被利用。虽然如此，RM算法还是一种最优的固定优先级调度算法，它在最差情况下的调度范围公式4.1如下：

$$W_n = n(2^{\frac{1}{n}} - 1) \tag{4.1}$$

其中，n为系统中任务的个数。

从以上公式可以看出，当系统中只有一个任务时，最差情况可调度范围是100%，但是随着任务数的增多，可调度范围随之降低，最终达到它的极限值69.3%(1n2)左右。所以可以得出：当系统中所有任务的总利用量不超过$n(2^{1/n}-1)$时，RM算法将可以调度所有的任务满足它们各自的时间要求。但这是充分不必要条件，在实际系统中，总有任务集具有较大并超过此限制的总利用量。令：

$$W_i = \sum_{j=1}^{i} e_j \lceil \frac{t}{P_i} \rceil \tag{4.2}$$

$$L_i(t) = \frac{W_i(t)}{t} \tag{4.3}$$

$$L_i = \min\{L_i(t)\}(0 < t \le P_i) \tag{4.4}$$

$$L = \max\{L_i\} \tag{4.5}$$

其中，$W_i(t)$是被执行任务T_1，T_2，…T_i，所执行的工作总量，在区间[0，t]被初始化；e_i是任务T_i的执行时间；P_i是任务的周期。

能够使RM算法可调度的充分必要条件是：$L_i \le 1$。

4.2.3 偶发性任务处理

非周期性的偶发性任务都是不规则地释放的，常常是对操作环境里发生的某个事件的反应。然而偶发性任务与这些事件没有周期性的联系，不过偶发性的任务必然存在有某种最大的释放比率。也就是说，在重复发生的偶发性任务成功释

放的间隔中，必然有某个最小的间隔时间。否则，能够添加到系统中的偶发性任务的工作总量将没有限制，这将无法确保满足任务完成的时间限的要求[28]。

一种处理偶发性任务的方法是：设定一个虚拟的周期性任务，此任务有最高的优先级并且有某个已经调好的假定的执行周期。在这段时间内，此任务将按照预定在处理器上运行，处理器能够被用来处理任何可能正在等候服务的偶发性任务。过了这段时间以后，处理器将专门处理周期性任务。

另一种处理偶发性任务的方法是：采用延迟服务的方法。无论什么时候，如果处理器被安排处理偶发性事件时，发现没有正在等候处理的偶发性任务，则处理器将按照优先级的顺序开始执行别的周期性的任务。然而，如果有一个偶发性的任务到来，此任务将抢占周期性任务的处理时间，最多能够占用的时间总数为分配给偶发性任务的时间。与基本RM算法的调度准则的推导类似，可以推导出DS算法的调度原理。当所有任务对应的时间限等于它们的周期，并且U_s是分配给偶发性任务的处理器使用权的时候，可以证明，如果总的任务利用(包括偶发性任务的贡献在内)U满足下面公式4.6的约束条件：

$$U \leq \begin{cases} 1-U_s (U_s \leq 0.5) \\ U_s (U_s > 0.5) \end{cases} \tag{4.6}$$

则调度周期性的任务集是有可能的。当$U_s \leqslant 0.5$的时候，有可能构造一个其调度不可行的周期性任务集，这些任务的处理器占用率为正任意小。

4.2.4 瞬间过载

RM算法的优先级只由其周期确定，这样就使得一些任务虽然实时性要求比较高，但是由于其周期数决定了它不能有一个高的优先级，所以必须合理地改变一些任务的优先级使得系统调度满足这些紧急任务的需要。动态改变任务周期是解决这一问题的比较好的办法。现在问题是如何调整任务周期数，使得紧急任务可以在最坏情况下得到实时性满足，一般的，可以通过减少紧急任务的周期数或者增加非紧急任务周期数来解决，编程过程示意图如图4.1所示。

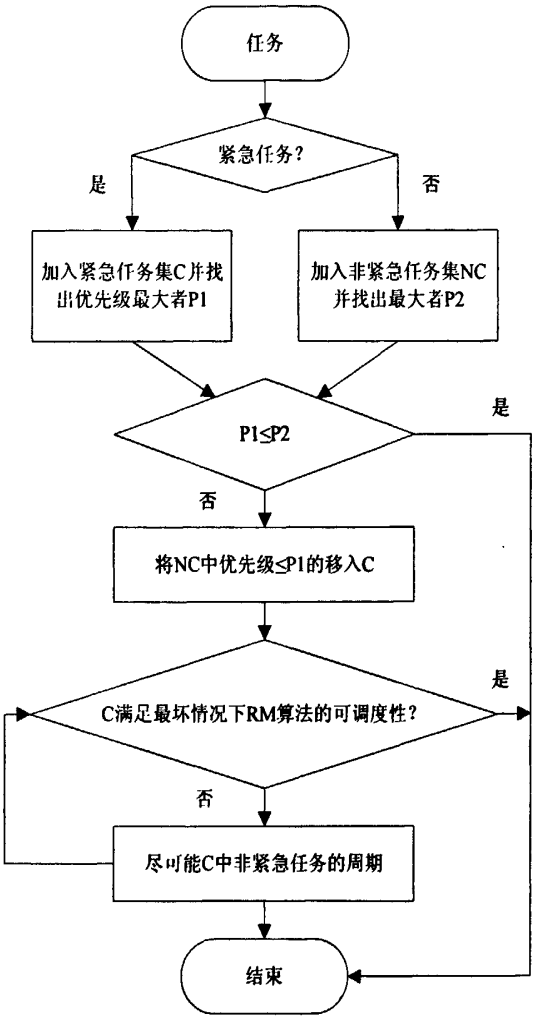

图 4.1 动态改变周期数编程示意图

4.3 EDF 算法

4.3.1 算法概述

EDF算法是一种可抢占的动态优先级调度算法，EDF算法按照任务的最终响应时限来调度，哪个任务的时限最早，就是最紧急的任务，就被赋予高优先级，得到优先调度。EDF算法中任务优先级在其初始时并不固定，而根据它们的绝对最后期限改变，所以遵照EDF算法调度的处理器总是执行所有任务中最早达到绝对最后期限的任务。

在讨论EDF算法时，将仍然保留在讨论RM算法时的假设，除了这一假设：所有任务都是周期性的。EDF算法是一种最优的单处理器动态调度算法，也就是说，假如EDF算法在单处理器上对一个任务集不能合理调度，那么其他算法也不能够对该任务集合理调度[29]。

EDF调度算法的实现步骤如下：

(1) 检查任务队列中所有就绪任务。

(2) 比较所有任务的最后时间期限。

(3) 将具有最先时间期限的任务给予最高优先级即最先执行该任务。

EDF算法的吸引力在于效率高、容易计算和推断，这是动态优先级调度研究的一个主要组成部分。EDF的缺点在于，理论表明这种算法能对可调度负载进行优化，但是它不能解决过载问题。发生过载时，导致CPU时间大量花费在调度上，性能退化很快。

EDF调度的实现相对容易。执行队列总是由下一个时限来分类。当一个任务处于激活状态时必须将该任务加入到执行队列中，或者如果任务的时限在当前正在执行的任务的时限之前，那么该任务就会抢先占用当前的任务。如果所有的任务都是周期性的(这些任务的出现仅仅由于时间的流逝)，那么抢先占用就没有必要。这样就简化了调度程序并且降低了任务切换的开销。如果任务可以在周期中的任何时刻执行，那么可实现性分析就简单可行。也就是说，早期的结果是可以接受的，并且任务准备好随时执行。从另一个角度来看，这样就允许任务的执行可以根据周期小幅变化。

EDF算法在实际应用中也存在一些问题，例如动态调度系统开销太大、在过载情况下会出现不稳定现象、优先级反转等。

4.3.2 非周期性任务的调度

调度程序的功能是调度一切任务，但是也有例外，像中断服务、DMA以及某些任务由于某些原因必须调度比原定执行序列更高的优先级，因而破坏原有的执行顺序。这些都称之为非调度的实体，且在时限调度程序中(如EDF)是最不受欢迎的。一些分析技术允许非调度实体的存在，但是却会导致调度程序的效率下降[1]。为时限调度程序而设计的系统总是尽可能地减少非调度实体的数量以及这些实体所占用的CPU时间。

非周期性任务对动态优先级调度程序来说是一个挑战，因为：

(1) 可实现性分析通常是一个必不可少的任务。如果系统需要尽可能地满足时限的要求，那么可实现性分析就不适合。

(2) 如果一个事件已经到达，而这时可实现性分析的结果表明不应该为这个任务提供服务。

(3) 非周期性事件的负载通常都描述为一种统计分布。

非周期任务事件可以使用比动态优先级任务更高优先级来调度[1]。这种方法有效的前提是非周期性事件需要硬实时服务并且周期性事件相对来说是软实时。主

要的问题是调度程序必须将所有的非周期性活动视为不可调度的实体，这样就使得在调度周期性任务时效率低下，这是因为存在大量不可调度的任务的缘故。经由动态优先级调度程序调度的周期性服务器能够为非周期性事件提供服务。这种方法将非周期性事件置于调度程序的控制之下。如何给非周期性事件提供较快的反应时间而不会为此而占用太多处理器时间。为了给非周期性事件提供与优先级调度系统中近乎一样的反应时间，服务器必须具有很短的调度周期。为了处理某一瞬间同时到达的非周期性事件的峰值负载，服务器为此而付出的代价就会很高，所以有时需要占用所有的处理器时间来处理这些非周期性的事件。

如果可以将非周期性事件分类为软实时，那么调度程序就可以更好地处理这些非周期性事件。非周期性服务器在每一个周期可以保留比平均资源开销略微多一点的系统资源而不是占用最大的系统资源。

4.3.3 优先级翻转与优先级继承

以上我们的谈论基于所有任务在任何时候可以被高优先级任务抢占的假设，但是有时任务必须对一些不可共享资源进行操作，所以当占用该资源的任务还未结束前，其他任务不能对其进行读写操作。实时系统中与资源相关联的任务占用不可共享资源时被称为在临界区。一种保证独占性资源访问的方法是利用二进制信号来看守临界区。当任务访问未被占用的临界区时，它会锁定与该临界区相对应的信号，当其他任务想访问该临界区时，首先需要察看该信号，如果信号锁定，则该临界区不能访问。当独占该临界区的任务完成访问并离开时，需要解锁该信号。同时，由于临界区而可能导致任务切换会产生问题。

例如当前系统中有两个任务，T_1，T_2。T_1优先级低于T_2，假如T_1首先运行并占用一个临界区，此时如果产生任务T_2而且它也需要访问该临界区，那么低优先级的任务T_1就阻塞了高优先级任务T_2的运行。虽然T_1优先级低，但是它会优于高优先级的任务运行，这种低优先级任务阻塞高优先级任务的现象被称为优先级反转。优先级反转会对系统多任务调度产生非常负面的作用。优先级反转问题可以通过优先级继承来解决。

当高优先级的任务被低优先级的任务由于临界区资源而阻塞运行时，低优先级任务可以暂时继承高优先级任务的优先数，当阻塞结束后，此任务再恢复他以前的优先级数，这种策略被称为优先级继承。优先级继承协议如下：

(1) 最商优先级任务T占用处理器，当它想访问一临界区并锁定该信号而该临界区已被其他任务占用时，T会放弃处理器。

(2) 假如由于争夺临界区任务T_1被任务T_2阻塞而且$T_1 > T_2$，那么在阻塞期间T_2将继承T_1的优先级数。当T_2离开该临界区时，它将恢复继承前的本身优先级。优先

级继承操作和以前优先级恢复操作是不可分割的两个步骤。

(3) 优先级继承是可传递的。假如T_3阻塞T_2，而T_2阻塞T_1(优先级$T_1 > T_2 > T_3$)，那么T_3将通过T_2继承T_1的优先数。

(4) 假如T_1没有被阻塞而且T_1的当前优先级高于T_2的当前优先级，那么任务T_1可以强占任务T_2。

以上讨论了优先级继承如何解决优先级反转问题，但是同时我们也需要注意到优先级继承协议的缺点：死锁和最高优先级任务阻塞。为此需要引入另外一个协议：优先级最高限协议，该协议可以解决这两个问题[1]。

优先级最高限是针对临界区的信号而定义的。一个信号的优先级最高限等于在可以锁定它的所有任务中的那个最大值的任务优先级数。优先级最高限协议基本与优先级继承协议相同，不同之处在于：当临界区S中当前存在其他任务T_2而且信号被其锁定，信号最高限大于或等于任务T_1优先级时，也许$T_2 < T_1$，但是此时T_1也不能抢占S。

优先级最高限协议具有的两个关键性质：

(1) 该算法不会产生死锁。

(2) 假设$B(i)$为所有可以引起任务T阻塞的临界区集合，$t(x)$为x区的被占用执行时间。那么，任务T可能被阻塞的最大值是$\mathrm{Max}_{x \in B(i)} t(x)$。通过性质2使得可以使用优先级最高限协议来推导系统可调度性分析。经过以上的一系列问题的讨论，可以得出结论：在RM调度算法中，当任务满足公式4.7时，任务T_i可以被调度满足其要求时限.其中$b_i = \mathrm{Max}_{x \in B(i)} t(x)$。

$$\frac{e_1}{T_1} + \frac{e_2}{T_2} + \ldots + \frac{e_i}{T_i} \le i(2^{\frac{1}{i}} - 1) \qquad (4.7)$$

4.4 本章小结

本章首先介绍了实时调度的基本知识，包括任务的特点，任务之间的相关性，实时调度算法及其分类等等。接着，介绍了两种比较经典的实时调度算法，分析了其主要理论依据。对 RM 算法及 EDF 算法在实际应用过程中所遇到的问题进行了详细的分析，并研究了相应的处理办法。

第五章 通用调度框架的设计和改进

RTAI是Real-Time Application Interface的缩写。它开发了一组实时应用结口供为开发人员使用，极大地提高了开发实时应用程序的效率。RTAI和RT-Linux的设计思想是基本类似的。它在Linux内核之上定义了一个新的内核，并可以直接将实时任务作为可加载内核模块运行，实际上每一个实时任务就是一个可加载式核心模块。所谓可加载式核心模块是指那些可在运行中动态加载卸载的基本内核功能模块。RTAI和RT-Linux最大的不同地方在于它在Linux上定义了一组实时硬件抽象层[30](RTHAL，Real-Time Hardware Abstraction Layer)。

5.1 RTAI 的关键技术

5.1.1 RTAI 体系结构和技术简介

RTAI体系结构如图5.1所示。

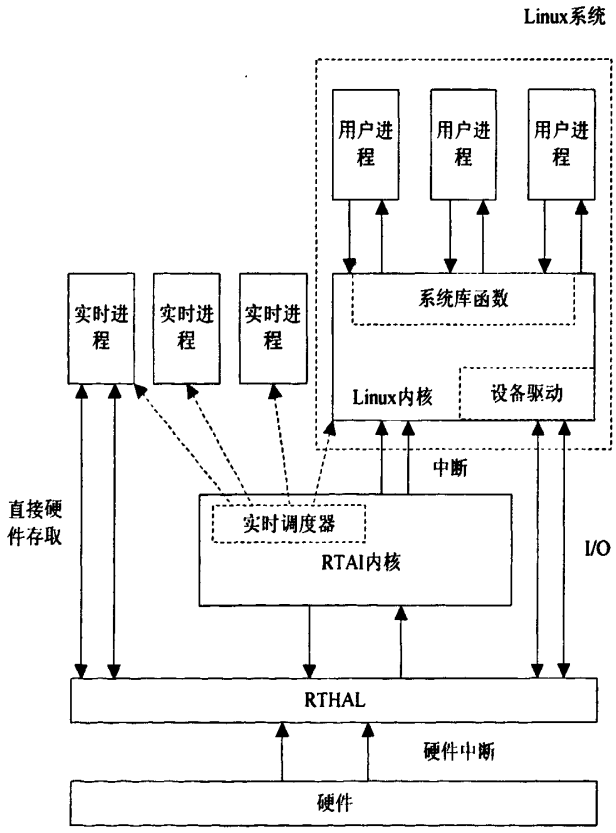

图 5.1 RTAI 体系结构图

RTAI的关键技术是通过软件来模拟硬件的中断控制器。当Linux系统要封锁CPU的中断时，RTAI中的实时子系统会截取到这个请求，把它记录下来，而实际上并不真正封锁硬件中断，这样就避免了由于封中断所造成的系统在一段时间没有响应的情况，从而提高了实时性。当有硬件中断到来时，RTAI截取该中断，并判断是否由实时子系统中的中断例程来处理还是传递给普通的Linux内核进行处理。具体实现机制如下[31]：

RTAI在Linux内核与硬件之间定义了一组实时硬件抽象层RTHAL，接管Linux对硬件平台的控制权。RTAI将那些与实时应用紧密相关的内核函数的指针(包括中断处理函数)和内部数据集中组织到rthal结构中。当系统要转入实时处理时，它们可以被RTAI动态转换。即在普通情况下，这些指针变量指向标准Linux中的函数，如开/关中断。此时系统相当于工作在标准Linux下，rthal不过增加了一个中间环节而已。当有实时要求时，即加载上RTAI的主模块后，该主模块的初始代码将这些指针变量修改为指向RTAI主模块中同名的函数，这时所有的硬件中断都将被实时硬件抽象层接收，由这些函数来处理。此时，当原有Linux试图屏蔽中断时，并没有真正屏蔽中断， 只是设置了rthal中的某个结构，也就是说实时内核仍可处理中断，而发生中断时，RTAI便不会调用Linux的中断处理程序，RTAI会首先接受中断请求，如果RTAI中存在对应的中断服务程序，则转至该服务程序，由实时内核的中断服务程序处理该中断。否则，查看rthal中的软件模拟中断控制器，如果Linux允许中断，则将中断交给Linux中断服务程序，如果不允许中断则返回。

5.1.2 RTAI 调度机制

在RTAI中，调度器实现了优先级调度。其实时任务控制块描述如下：

```
struct rt_task_struct
{
    struct rt_threadtss; //任务状态段
    unsigned long kernel stack;//系统堆栈
    struct rt_mm_informm; //内存分配信息
    int pid;//实时任务的任务号
    long state;//实时任务的状态
    int priority;//优先级
    Rtime resume_time;// 下次执行时间
    Rtime period;//周期
};
```

其中实时任务的状态可以是以下几种：

RT_TASK_STOPPED：实时任务已经停止。

RT_TASK_RUNNABLE：实时任务处于就绪状态，一旦获得CPU的控制权即可立即执行。

RT_TASK_WAITING：实时任务被阻塞。

所有的任务控制块组成一个单链表，便于统一管理所有实时进程：所有就绪任务的任务控制块按照优先级高低组成一个双向循环链表，便于调度的时候能快速选择最高优先级就绪任务进行调度；链表的头节点代表Linux。所有被等待时钟信号的阻塞进程按照等待时间的长短组成一个双向循环链表，以便于在one-shot模式下对时钟芯片进行设置。

系统初始化时，数组rt_task[RT_NR_TASKS]的所有表项被初始化为空表项。每次实时调度器在执行时，首先将阻塞进程链表扫描一遍，找出第一个resume_time小于下次时钟中断时间的任务，将下次时钟中断时间设置为resume_time。然后在所有就绪实时任务中找出优先级最高的实时任务，将其投入运行[31]。

从上述实时任务的调度分析中可以看出，RTAI的实时任务控制块以及调度程序比较简单而精巧，反映了其设计者的一个主导思想，将实时应用可以分为两个域，即实时域和非实时域。实时域只对应用中与时间密切相关的部分进行处理，而大量的复杂处理则留给非实时域处理。

RTAI的这种基于静态优先级的调度算法十分简单有效，能够有效的降低系统的调度延迟和上下文切换时间，提高系统的实时性。然而这种算法也有它的局限性[23]：

(1) 实时任务的优先级完全由开发人员确定。这样增加了开发人员的负担，对开发人员提出了更高的要求。实时任务的优先级安排是否合理，直接影响系统中的多个实时任务能否在规定时间内完成。

(2) 不支持动态优先级，这使得RTAI的调度缺乏灵活性，不能应用于适合动态优先级调度的场合。

为了解决RTAI在调度机制上存在的局限，本文对RTAI的调度器进行了修改，重新设计了一个通用调度策略的框架，扩展了其实时应用的范围，下面详述了系统的设计方案及实现。

5.2 实时调度框架设计与实现

5.2.1 系统框架设计

因为本文选择的实现平台是RTAI，它是双内核结构，因此整个系统也是基于双内核结构的。其中实时内核本身的代码具有良好的可抢占性，并且具有可快速响应的中断机制，有能力处理硬件中断。这个实时内核的实现虽然精简，但包括了基本的功能，如：任务管理机制、任务间通信机制、基本的优先级调度机制等，实时任务可在其上运行。与此同时，Linux内核及其上的应用进程集作为一个整体被看作运行在实时内核之上的一个任务，具体的系统总体方案如图5.2所示。

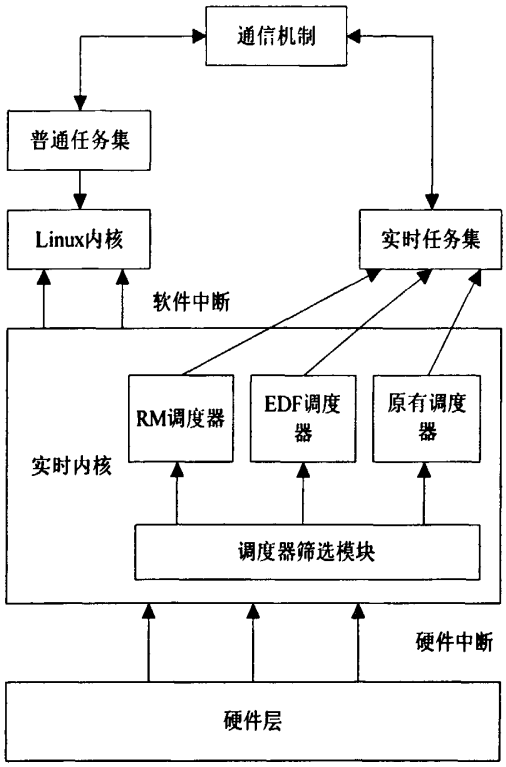

图 5.2 系统框架设计结构图

对于本文设计的通用调度策略框架而言，重要的是这里的实时内核具有良好的可抢占性，使其可以很好地支持RM静态调度策略和EDF动态调度策略。整个调度框架是在实时内核的地址空间中实现的，这样，为实现通用调度策略框架而编写的各个子程序可以作为实时内核的一部分运行。由于在RTAI中，实时内核之上运行的各个实时任务是以动态加载的模块形式存在的，当它们装载到内核中后，本质上就是内核的一部份。具体来说，RTAI是模块化的，当被装载的时候，模块代码放在内核地址空间，在内核的上下文中运行，要在提供两个函数情况下才能工作，这两个函数是init_module()和cleanup_module()。 init_module()在模块被装入时执行，而cleanup_module()在卸载时使用。

5.2.2 系统调度主模块

要对RTAI的实时调度器进行改进，就必须要对调度器所涉及的主要部分进行深入的分析。调度器所涉及的模块主要有调度模块和定时器模块，这几部分的源程序主要在sched_up.c，comman.c以及相应的头文件中。

实时任务通过rt_task_init()函数创建并进行初始化，为任务创建相应的任务控制数据结构并分配堆栈。初始化完成后任务处于挂起状态，用户可通过调用rt_task_resume()函数或者rt_tast_make_periodic()函数激活该任务。系统通过rt_schedule()函数来对任务进行调度，选择出最高优先级的任务，调用rt_switch_to()

函数进行任务切换，产生任务的上下文切换的堆栈操作，从而让被选中的实时任务得以执行。实时任务在执行的过程中可以调用rt_task_suspend()函数将任务挂起，通过调用rt_task_resume()将任务唤醒；通过task_wait_period()将任务挂起等待下一周期的到来，通过rt_task_yield()函数停止任务的本次执行等等。任务状态的每次变化都会使系统调用rt_schedule()函数重新引起调度。

调度器通过start_rt_timer()函数启动实时调度定时器，rt_request_timer()函数将为定时器安装上默认的定时器中断处理程序rt_timer_handler()。定时器中断时间一到即触发中断，系统立即调用默认的定时器中断处理函数rt_timer_handler()，该函数的主要功能是设置下次中断时间，管理任务的运行时间，引起系统的任务调度。

由以上的分析可以得出，RTAI调度器中引起系统调度的时机有两个，一个是系统时钟中断到来时进行调度，一个是任务状态变化时调用系统调度函数进行调度。相关的两个主要函数分别是系统中断处理函数rt_timer_handler()和系统的调度函数rt_schedule()。RTAI的实时任务调度就在这两个函数中完成任务，本文要实现的调度算法就主要是针对这两个函数进行的。

5.2.3 添加调度策略

(1) RM调度器中主要的数据结构的改造

首先，为了在RTAI中添加RM调度策略，必须在头文件sched.h中增加RM算法的宏定义：

```
#define SCHED_RM 1
```

另外，由于RTAI的进程描述结构rt_task_struct里有表示任务周期的项Rtime period，因此这里不需要对数据结构rt_task_struct进行修改。

(2) EDF调度器中主要数据结构的改造

首先，为了在RTAI中添加EDF调度策略，必须在头文件sched.h中增加EDF算法的宏定义：

```
#define SCHED_EDF 2
```

另外，对任务控制块数据结构rt_task_struct进行修改，添加两个新的项Rtime deadline和Rtime deadline_cur，分别表示任务的初始截止时间期限和当前的截止时间期限。然后提供一个函数接口rtai_set_task_deadline()，让用户可以对任务的初始截止时间期限deadline进行设置。

```
void rtai_set_task_deadline(&rt_task_struct task,Rtime time)
{
    task->deadline=time;
    task->deadline_cur=time;
}
```

5.2.4 增加调度算法

在RTAI实时内核中，共定义了三种与任务相关的队列，它们分别是：任务队列、延时队列和就绪队列。这些队列在实现时都是采用双向链表形式，以方便快速的查找。在实时任务的结构体中有它们的指针定义：

```
typedef struct rt_struct{
    struct rt_task_struct *next;//任务队列前指针
    struct rt_task_struct *tprev;//等待队列前指针
    struct rt_task_struct *tnetx;//等待队列后指针
    struct rt_task_struct *rprev;//就绪队列前指针
    struct rt_task_struct *rnetx;//就绪队列后指针
    ……
}
```

队列管理主要涉及的数据结构就是以上的三种链表，其中任务队列中存放所有系统中已创建并且未删除的任务；延时队列中存放在当前时刻还无权竞争处理机的任务，对它的主要操作为插入(按照起始时间渐增顺序)和删除：就绪队列中存放可竞争处理机执行权的就绪态任务，对它的主要操作为插入和删除。在实时内核中，这部分的一些相关函数如下：

```
enq_timed_task();//将任务插入延时队列
rem_timed_task();//从等待队列删除任务
rem_ready_task();//从就绪队列删除任务
```

为了实现RM算法和EDF算法调度功能，需要增加两个函数，enq_rm_task()和enq_edf_task()，他们分别将任务按照RM算法和EDF算法插入到就绪队列中。下面简要介绍一下这两个函数。

(1) enq_rm_task()

这个函数是将任务按RM算法插入到就绪的队列中。其主要实现如下：

```
static inline void enq_rm_task(RT_TASK * ready_task)
{
    RT_TASK *task;
//按固定优先级渐增顺序查找插入点
    while(ready_task->priority>=priority)
    {
        if((task=task->next)->priority<0)
            break;
    }
    //执行插入操作
    task->rprev=(ready_task->rprev)->rnext=ready_task;
    ready_task->rnext=task;
}
```

(2) enq_edf_task()

这个函数是将任务按EDF算法插入到就绪的队列中。其主要实现如下：

```
static inline void enq_edf_task(RT_TASK *ready_task)
{
    RT_TASK *task;
    //按截止期渐增顺序查找插入点
    While(task->policy&&ready_task->period=task->period)
```

```
    {
        task=task->next;
    }
    //执行插入操作
    task->rprev=(ready_task->rprev=task->rprev)->rnext=ready_task;
    ready_task->rnext=task;
}
```

5.2.5 调度选择器的实现

首先，增加一个全局变量rtai_schedule_alg，用这个全局变量来标识系统当前的调度算法。这个变量只在实时任务模块被加载的时候修改，即在模块的init_module()中被调用，不会有两个任务同时试图修改此变量，因此不需要利用自旋锁等进行互斥操作。

```
int rtai_schedule_alg=0;
#define SCHED_ORIGINAL 0 //系统原有的调度算法
#define SCHED_RM 1 //增加的RM调度算法
#define SCHED_EDF 2 //增加的EDF调度算法
```

接下来需要增加一个函数rtai_make_schedule_alg(int schedule_alg)，用来向用户提供接口，用户通过这个函数来选择适用的调度算法。

```
int rtai_make_schedule_alg(int schedule_alg)
{
    int last_alg;
    if(schedule_alg!=0&&schedule_alg!=1&&schedule_alg!=2)
        return -1;
    last_alg=rtai_schedule_alg;
    rtai_schedule_alg=schedule_alg;
    return last_alg;
}
```

在初始状态下，系统默认rtai_schedule_alg的值为0，即RTAI自带的静态优先级调度算法。如果用户需要选择其他的调度算法，就需要调用rtai_make_schedule_alg进行修改设置。

最后，对RTAI的调度函数rtai_schedule()进行修改：

(1) 增加三个调度子函数：

```
int rtai_schedule_original(void)
int rtai_schedule_rm(void)
int rtai_schedule_edf(void)
```

(2) 让rtai_schedule_orignal()执行RTAI原有调度函数rtai_schedule()的功能。

(3) 将原来的调度函数rtai_schedule()改造为一个调度算法选择器，其主要代码如下：

```
switch(rtai_schedule_alg)
{
    case SCHED_ORIGINAL:
        return rtai_schedule_original();
        break;
    case SCHED_RM:
        return rtai_shcedule_rm();
```

```
        break;
        case SCHED_EDF:
            return rtai_shcedule_edf();
            break;
        default:
            return -1;
    }
```

调度算法选择器的流程图如图5.3所示。

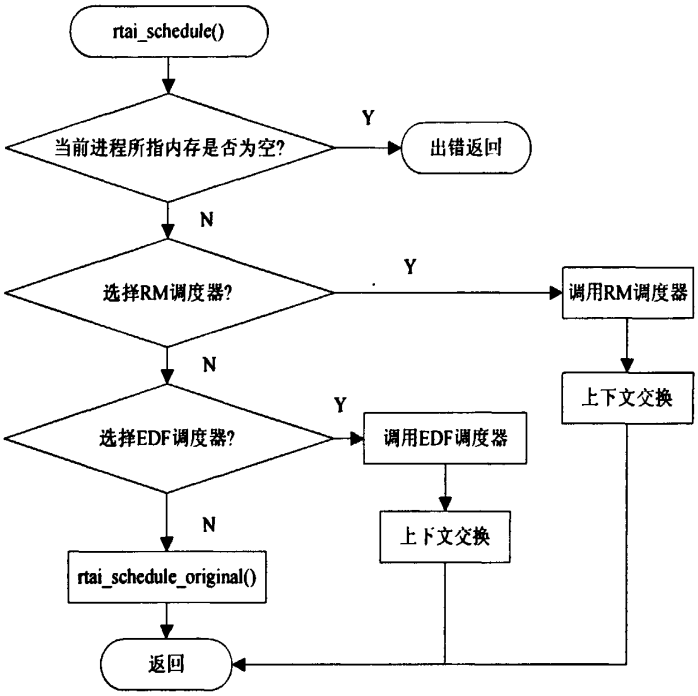

图 5.3 调度算法选择器的流程图

5.3 调度器设计

任务调度策略是直接影响实时性能的因素。尽管调度算法多种多样，但大多由单调速率算法RM和最早时限优先算法EDF变化而来。前者主要用于静态周期任务的调度，而后者主要用于动态调度，在不同的系统状态下两种算法各有优劣。

5.3.1 RM 调度器的设计

RM调度算法是静态优先级调度常用的实时调度算法。本文将设计RM调度器。

首先假设RM调度算法调度的任务具有如下特点：

(1) 在单个CPU中，所有进程都是周期性运行的；

(2) 进程间没有数据依赖关系，即各个进程相互独立，其完成和启动不依赖于其他进程的完成与启动；

(3) 进程的执行时间是恒定的：

(4) 所有进程的最后期限都在它周期的结束点上；

(5) 优先级最高的就绪进程会先被选择执行。

针对以上特点，设计RM调度器，要完成的工作包括以下几个：

(1) 优先级的扩展。由于RM算法最终是将任务的周期映射成优先级，在运行队列中等待调度，等待调度的进程根据优先级选择合适的进程调度，因此需要把RM任务的周期映射为合适的优先级。Linux实时任务优先级可以从0-99，所以可以将RM任务的优先级映射到实时任务优先级之上，映射是采用线性函数。为此，需要定义RM任务所允许的最大周期间隔和最小周期间隔。增加宏定义：

```
#define MAXP
#define MINP
```

MAXP可以取为50，MINP取1即可。

设计RM优先级为(100，100+MAXP)之间的数，因此线性函数定义为：

Priority=100+(MAXP-peirod)

此线性函数的优点在于既能满足周期短优先级高，周期长优先级低的要求，也使得计算出的优先级一定为整数，映射关系如图5.4所示。

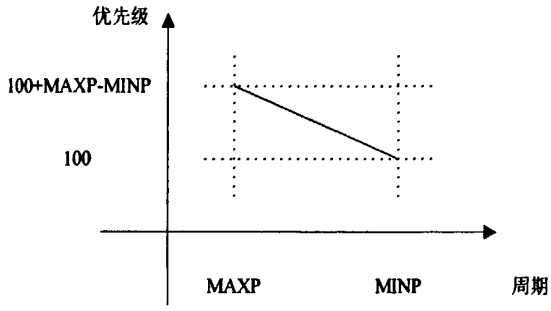

图 5.4 RM 中周期到优先级的映射关系

(2)RM调度函数rtai_schedule_rm()

rtai_schedule_rm处理过程如下：

①RM调度开始，将指针new_task指向任务链的首部；

②判断时钟中断是否为oneshot模式，若不是则遍历可以运行的任务链，指针new_task指向周期period最短的任务，并转向⑤；

③遍历可以运行的任务链，指针new_task指向周期period最短的任务；

④遍历休眠任务链，如果某个任务的恢复运行时间小于下次中断时间，则将下次时钟中断时间调整为这个任务的恢复运行时间；

⑤判断new_task所指向的是否为当前任务，若是则运行当前任务并结束；

⑥如果当前任务为Linux，则保存Linux进程的上下文；

⑦如果new_task为Linux，则恢复Linux进程的上下文；

⑧调用“rt_switch_to(new_task)函数切换到new_task，运行new_task所指的任务

并结束。

RM调度器的整个处理流程如图5.5所示。

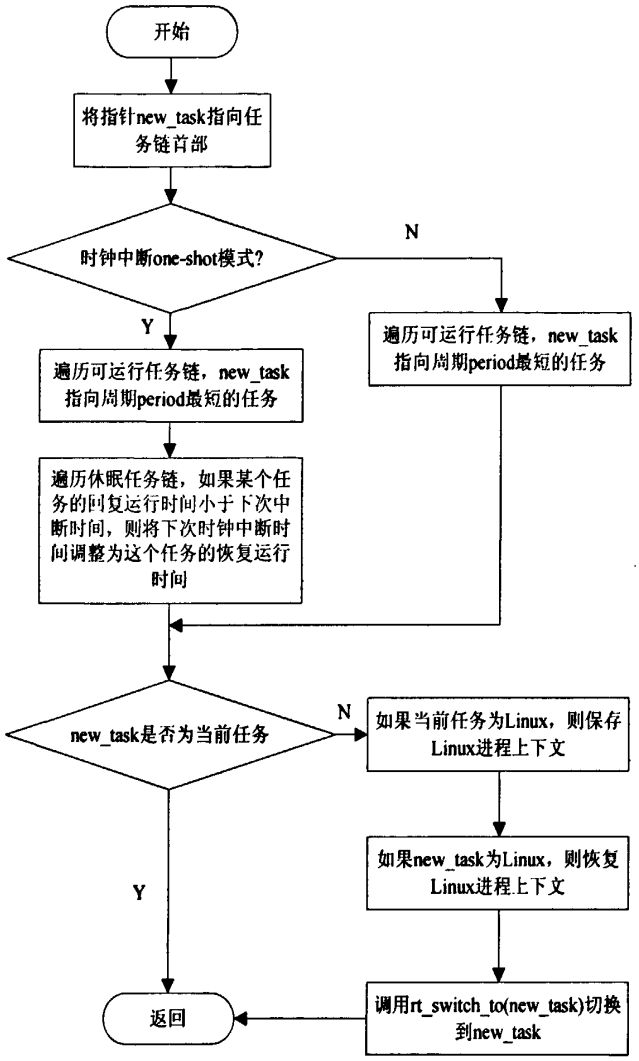

图 5.5 RM 调度器的处理流程图

5.3.2 EDF 调度器的设计

EDF调度算法是动态优先级调度常用的实时调度算法。本文将设计EDF调度器，要完成的工作包括以下几个：

EDF调度函数rtai_schedule_edf()

rtai_schedule_edf()函数的具体工作流程跟rtai_schedule_rm()大体上相同，不同的地方在于，当遍历可运行任务链时，让new_task始终指向deadline_cur最小的任务，调度器最终选中的就是具有最小deadline_cur的任务。另外，EDF是一种动态优先级调度算法，可运行任务的deadline_cur需要在时钟中断处理时进行更新，即deadline_cur减去上一个时钟中断间隔时间作为新的deadline_now，而调度器则需要

将换出的任务的deadline_cur恢复成初始值deadline。

rtai_schedule_edf()处理过程如下：

①EDF调度开始，将指针new_task指向任务链的首部；

②判断时钟中断是否为oneshot模式，若不是则遍历可以运行的任务链，指针new_task指向当前截止期最短的任务，并转向⑤；

③遍历可以运行的任务链，指针new_task指向当前截止期最短的任务；

④遍历休眠任务链，如果某个任务的恢复运行时间小于下次中断时间，则将下次 时钟中断时间调整为这个任务的恢复运行时间；

⑤运行new_task所指向的任务；

⑥判断当前时间片是否用完，是则对每个进程调用update_deadline()更新其当前截止期，并转向①；

⑦继续判断任务是否执行完毕，是则换出任务，更新其当前截止期为初始截止期，结束，否则转向⑤。

EDF调度器的整个处理流程如图5.6所示。

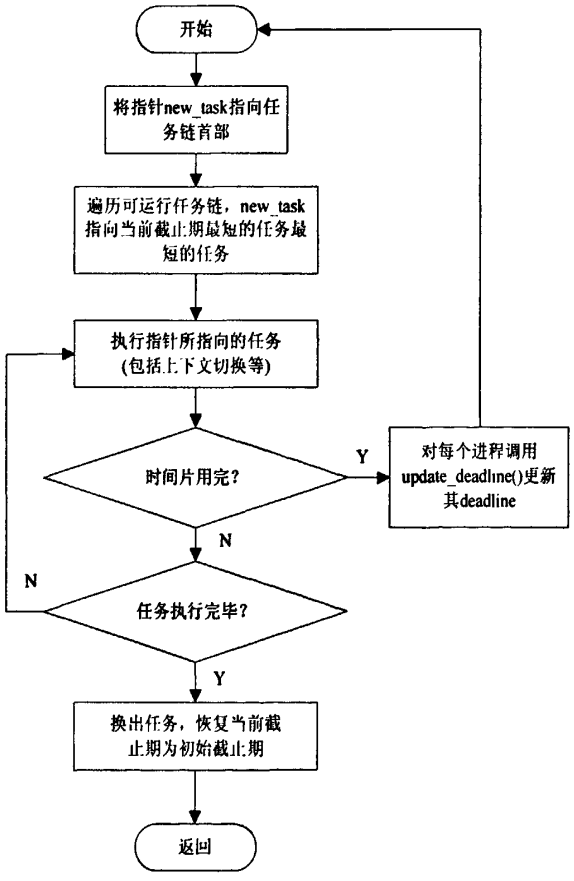

图 5.6 EDF 调度器的处理流程图

rtai_schedule_edf函数调用update_deadline(p)函数来动态更新任务的优先级。update_deadline(p)函数首先比较系统jiffies与当前进程deadline。若系统jiffies大于当前进程的deadline则更新deadline，执行deadline+=period，然后继续判断并执行一直

到deadline大于或等于jiffies为止。通过执行update deadline(p)函数使得系统进程的最后时限动态的更新，保证rtai_schedule_edf()在下次调度的时候可以执行最后时限最短的进程，从而使得EDF算法得以有效的调度。

5.4 实验结果分析

5.4.1 实验设备和测试前的准备工作

(1) 实验环境

本文所有的试验的硬件环境为单处理机的兼容PC，主要部件是：AMD Sempron(tm) Processor 2800+ 1.60GHz，512MB内存，80G硬盘。软件环境为：Fedora8.0+rtai3.6。

(2) RTAI的编译安装过程

RTAI的编译安装包括以下几个步骤：

①把rtai3.6.tar.gz解压缩到/usr/src目录下

②建立链接

#cd/usr/src

#ln -sf rtai-3.6 rtai

③将改进后的调度模块添加到shed.c中，并Patch实时内核

#cd/usr/src/linux

#patch -pl< /usr/src/rtai/base/arch/i386/patches/hal-linux-2.6.23_rx.patch

④配置编译内核

#cp /boot/config-2.6.23*fc8 .config

make oldconfig

#make menuconfig(设置或者取消一些选项)

#make modules

#make modules_install

⑤引导实时内核

最后重新启动机器：#reboot

⑥安装RTAI实时模块

#cd/usr/src/rtai

#make menuconfig(设置或者取消一些实时选项)

#make

#make modules_install

5.4.2 系统测试

接下来将对改进后的RTAI进行测试，看实时任务是否按照被设置的参数被调度。首先，在一个模块中创建10个周期性实时任务task0-task9，赋予它们相同的优先级，不同的周期，不同的截止期。task0-task9的周期分别设置为800μs到809μs，以1μs的速率递增。相反，截止期分别设置为809μs到800μs，以1μs的速率递减。这些实时任务并不执行实际功能，而是与一个Linux下的进程PrintResult进行通信。当第i个实时任务被调度器选择获得运行机会的时候，这个任务首先将字符串，"i began"通过FIFO发送给Linux下的接收进程PrintResult，表示任务i开始运行，然后执行10000次的递加操作，消耗一定的运行时间后，将字符串"I waited"通过FIFO发送给PrintResult，最后进入周期性休眠状态，整个系统的测试程序见附录。任务1,2...,10的主函数跟test0()类似，只是输出到FIFO的字符不同。模块的初始化函数为在加载模块时被调用，模块的清除函数在卸载模块时被调用。

为了能及时显示测试结果，Linux下的进程PrintResult专门用于接收实时任务通过FIFO传递的信息，再将这些信息在终端上进行打印输出。这也是RTAI的一种典型应用方式，即实时任务进行实时数据的采集并传递给Linux进程，Linux进程利用Linux丰富的系统调用，来处理和显示这些数据。

测试结果1：当调度算法设置为SCHED_RM时，随机抽取PrintResult的输出结果如图5.7所示。

```
task0 began
task0 waited
task1 began
task1 waited
task2 began
task2 waited
task3 began
task3 waited
task4 began
task4 waited
task5 began
task5 waited
task6 began
task6 waited
task7 began
task7 waited
task8 began
task8 waited
task9 began
task9 waited
task0 began
task0 waited
task1 began
task1 waited
...
```

图 5.7 RM调度算法测试结果

可以看出，调度情况完全符合预想情况，即周期相对小的任务先于周期长的任务获得运行。

测试结果2：当调度算法设置为SCHED_EDF时，随机抽取PrintResult的输出结果如图5.8所示。

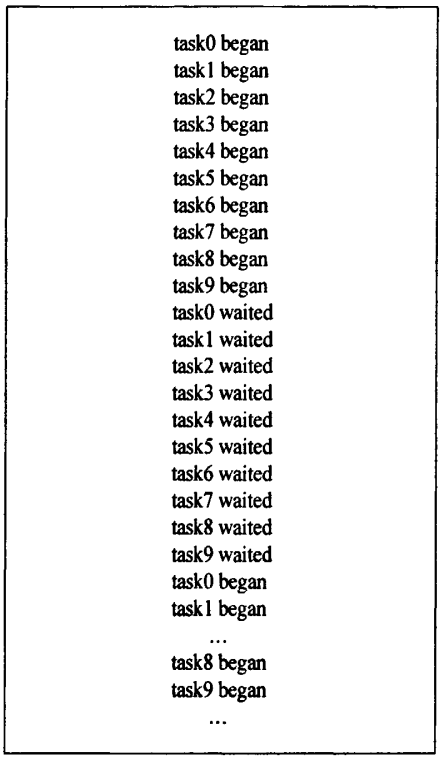

图5.8 EDF调度算法测试结果

从结果可以看出，任务运行过程中频繁的发生抢占式调度。其原因是随着系统的运行，时间的推移，未运行的任务的deadline在不断减小。当某一任务的deadline在某一时刻变得比正在运行任务的deadline短时，就会发生抢占，如task0还未结束，task1就获得了CPU，得以运行。试验结果与预期情况吻合，表明调度过程符合EDF调度算法思想。

以上两个测试验证了经过调度改进后的RTAI能够正常运行，能依照所设定的算法对实时任务进行正确调度。

操作系统的实时性能可以由任务响应时间、调度延迟时间、上下文切换时间等几个重要参数来衡量，其中最主要的指标是任务响应时间，这是一个衡量实时操作系统整体实时性能的指标，也是一个非常直观又有实际意义的指标。需要注意的是，操作系统的实时性能不仅仅取决于操作系统本身，还受硬件平台的影响，而且用来获得性能数据的测试环境的不同也会使得测试结果略有差异。

测试结果3：调度算法设置为SCHED_RM时，任务平均响应时间测试如表5.1所示。

表 5.1 平均响应时间测试结果

任务平均响应时间(ns)	超时运行进程数
289	0
267	0
314	0
350	0
332	0
274	0
286	0
270	0
273	0
272	0
264	0
265	0
263	0
265	0
265	0
265	0
265	0
264	0
264	0

表中第一列表示任务平均响应时间，单位是纳秒，第二列表示的是任务超时的总数，上面的测试表明改进后的系统具有很小的任务平均响应时间，并且没有超时的任务发生。

测试结果4：当调度算法设置为SCHED_EDF时任务响应时间测试如表5.2所示。

图中第一列是最小调度延时，第二列是平均调度延时，第三列是最大调度延时，测试表明改进后的系统具有很小的平均调度延时。

表 5.2 平均调度时间测试结果

最小调度延时时间(ns)	平均调度延时时间(ns)	最大调度延时时间(ns)
237	281	3770
237	288	3770
237	275	3770
237	292	3770
237	272	3770
237	278	3770
237	294	3936
237	279	3936
237	280	3936
237	278	3936
237	279	3936
237	291	3936
237	275	3936
237	279	3936
237	289	3936
237	277	3936
237	281	3936
237	283	3936
237	278	3936

测试结果5：系统总体性能测试，上下文切换时间如表5.3所示。

表 5.3 上下文时间切换时间测试结果

	切换总次数	总用时(ms)	平均用时(ns)
进程挂起和唤醒之间的切换	40000	27	695
发送和等待进程之间的切换	40000	28	707
发送和接受进程之间的切换	40000	38	941

本实验10个进程之间循环切换，第一个测试是进程挂起和唤醒的切换时间，10个进程循环进行40000次切换，共用时27ms，平均切换时间为695ns，第二个测试是信号量的发送进程和等待进程间的切换时间，共用时28ms，平均切换时间为707ns，第三个测试是进程间通信时发送进程和接收进程的切换时间，共用时37ms，平均切换时间为941ns，结果验证了改进后的系统具有良好的实时性能。

5.5 本章小结

本章主要介绍的是在RTAI基础之上的调度框架的设计和实现，是本文的核心内容。首先简单介绍了RTAI的关键技术和调度机制，通过分析可以知道RTAI具有良好的硬实时性，但是其实时调度机制比较单一，造成应用范围有限。受第三章介绍的RED-Linux的通用调度框架的启发，本文对RTAI的调度框架重新设计，使其具体多种调度策略。基于静态优先级的RM实时算法和动态优先级的EDF实时算法是比较经典的实时调度算法，在某些情况下也是最优的实时调度算法。因此，将RM和EDF调度算法添加到调度框架中去，从而扩大了它的应用范围，使得RTAI在应用上将更加灵活，适合嵌入式系统不断发展的要求。本章最后对实现的实时系统进行了功能和性能测试，结果表明改进后的实时系统具有良好的调度机制和性能。

第六章 结束语

6.1 工作总结

本文通过对Linux调度机制以及实时化技术的研究和分析，的讨论了基于Linux操作系统的实时性改造问题。RTAI是一种成功的Linux实时化方案，但是调度机制比较单一，可以改善其调度框架来扩展其应用。因此，本文主要完成的工作如下：

(1) 对Linux调度机制进行深入的分析，指出了Linux操作系统在实时应用上的不足之处，并总结了Linux实时化的主流技术，分析了几种流行的Linux实时化方案。

(2) 对调度算法进行了深入的研究，详细论述了两种比较经典的实时调度算法——RM算法和EDF算法，讨论了在实际应用过程中遇到的一些问题。

(3) 研究了RTAI的关键技术，指出了其优点以及不足之处，对RTAI的实时调度框架进行重新设计和改进，实现了经典的RM和EDF实时调度算法，将其添加到重新设计的RTAI调度机制中，使得RTAI的应用范围扩大，改进了其单一的调度机制。

(4) 对改进后的 RTAI 进行了功能测试和性能测试，结果表明经过改进后的RTAI 能够运行正常，并能根据指定的算法对实时任务进行正确的调度，性能测试表明 RTAI 仍然具有良好的实时性。

6.2 展望

本文仅仅从调度算法的角度对RTAI进行了扩展，所做的工作还处于初级阶段，基于Linux的操作系统还需要进一步的研究，主要问题如下：

(1) 本文所实现的调度算法能满足某些实时应用的需要，但是在嵌入式应用范围越来越广泛的情况下，增加对更多的实时调度算法的支持将使该系统获得更广泛的应用。

(2) 目前针对实时Linux的集成开发环境和调试工具还不多，实时应用开发不方便，调试难度比较大。

(3) 目前本系统只是针对单处理机系统设计的，而多处理机在未来的实时系统领域中会有广大的应用空间，因此，还需要研究在多处理机平台下的实时调度框架和算法。

(4) 本系统所做的工作主要集中在实时应用的扩展上，但是如果要将其真正运用到实际的嵌入式系统中去，还需要进行内核的剪裁和移植等工作，这两项工作

还是比较具有挑战性的，因为移植后的系统要足够的小，并且能够支持 RTAI。

(5) RM 和 EDF 调度算法在实际应用中会遇到很多问题，如优先级反转等，需要对其进一步优化，以达到实际应用的要求。

总结本文，本文所作的工作为开发嵌入式实时 Linux 提供了一定的基础，进一步的开发，可以现在的基础之上，围绕着上述讨论的方向继续进行。

致谢

衷心感谢我的指导老师曹伯燕教授！在两年多的研究生学习生涯中，曹老师在各方面给予了我极大的帮助。曹老师不但在对我的研究课题和专业进行了孜孜不倦的指引和教导，而且还为我们提供了很多实践的机会，锻炼了我的思考和解决问题的能力。这期间，曹老师深厚的学识功底，敏锐的学术洞察力，严谨治学作风，精益求精的工作态度，时时影响着我，同时曹老师的为人处世之道也给我留下很深的印象，这将对我以后的学习和工作产生深远的影响。至此学位论文完成之际，谨向敬爱的曹老师致以最诚挚的谢意。

衷心感谢崔巍、赵永刚、薛天培、杨爽、李慧路、刘文剑、曾丽娜、郭婧、王月梅和安妍君等同门师兄妹们，在与他们的讨论过程中，我的专业知识在不断增长，感谢他们在学习方面对我的帮助；在生活中，是他们和我一起度过了这两年多的美好时光，感谢他们在生活中对我的帮助。

衷心感谢我的父母，这么多年的生活学习是在他们的鼎力支持下完成的，因为他们无私的奉献，我顺利了完成学业，衷心感谢他们对我的养育、照顾和支持。

衷心感谢所有帮助过我的老师、同学、亲戚和朋友们，是他们让我的学习和生活变得更加多姿多彩。

参考文献

[1] Jane W.S.Liu. 实时系统. 姬孟洛，李军等译. 北京：高等教育出版社，2003

[2] Qing Li,Caroline Yao. 嵌入式系统的实时概念. 王安生译. 北京：北京航空航天大学出版社，2004

[3] C.M.Krishna,Kang G.Shin. 实时系统. 戴琼海译. 北京：清华大学出版社，2004

[4] 张焕强. 基于Linux的实时系统. http://www.ibm.com/developerworks/-cn/linux/embed/l-realtime/index.html, 2003

[5] 王继刚，顾国昌，徐立峰，李翌. 强实时性 Linux 内核的研究与设计. 系统工程与电子技术，2006，28(12)：1932-1934

[6] 朱珍民，隋雪青，段斌. 嵌入式实时操作系统及其应用开发. 北京：北京邮电大学出版社，2006

[7] 郑宗汉. 实时系统软件基础. 北京：清华大学出版社，2003

[8] 马毅. 基于 Linux 的实时操作系统设计. 计算机工程与应用，2001，(23)：109-11

[9] 赖娟. Linux 内核分析及实时性改造. 电子科技大学硕士学位论文，2007

[10] Daniel P.Bovet,Macro Cesat.Understanding the Linux kernel.O'Reilly,2002

[11] 杜旭，晋海鹏. Linux 操作系统调度器实时性能的研究和改进. 计算机工程，2005，31(10)：100-102

[12] 郭玉东. Linux 操作系统结构分析. 西安：西安电子科技大学出版社，2002

[13] 胡希明，毛德操. Linux 内核源代码情景分析(上). 浙江大学出版社，2001

[14] Robert Love. Linux 内核设计与实现. 陈莉君，康华等译. 北京：机械工业出版社，2004

[15] 林浒，蔡光起，李凤宪，钟利明，郭锐锋. 实时化的 Linux 系统及其实时性能的研究. 小型微型计算机系统，2004，(8)1454-1457

[16] 左天军，左园园. Linux 操作系统的实时化分析. 计算机科学，2004，31(5)：110-112

[17] 李江，戴胜华. Linux 操作系统实时性测试及分析. 计算机应用，2005，25(7)：1679-1681

[18] 廖永刚. 一种实时 Linux 操作系统—RTAI 的分析与扩展. 福建电脑，2005，(5)：75-76

[19] Lin KJ,Wang YC.The design and implementation of real-time schedulers in RED-Linux.Proceedings of the IEEE ,2003,91(7)

[20] 吴一民. RT-Linux 的实时机制分析. 计算机应用，2002，22(12)：100-111

[21] 须纹波，张星烨，欧爱辉. 基于 RTAI-Linux 的实时操作系统的分析与研究，现代计算机，2003：19-21

[22] 何军. 实时调度算法研究. 北京：中国社会科学院软件研究院，1997

[23] 邢群科. 进程实时调度算法研究. 北京科技大学硕士学位论文，2005

[24] 冯艳红，张玉明，徐美华. 实时调度算法分类研究. 微型电脑应用，2005，21(7)：12-13

[25] 翟鸿鸣. 单处理器系统的实时调度算法研究. 微机发展，2003，13(10)：99-10

[26] 王强，王宏安，金宏等. 实时系统中的非定期任务调度算法综述. 计算机研究与发展，2004，41(3) ：385-391

[27] 邢建生，刘军祥，王永吉. RM 及其扩展可调度性判定算法性能分析. 计算机研究与发展，2005，42(11)：2026-2030

[28] 罗玎玎，赵海，孙佩刚，张希元，尹震宇. RM 算法的运行时开销研究与算法改进. 通信学报，2008，29(2)：79-86

[29] 洪艳伟，赖娟，杨斌. 基于 EDF 算法的可行性判定及实现. 计算机技术与发展，2006，16(11)：97-99

[30] D.Beal,E.Bianchi,L.Dozio.RTAI:Real Time Application Interface.Linux Joumal, 2003(4)

[31] P.Mantegazza,E.Bianchi,L.Dozio,S.Papacharalambous,S.Hughes, D.Beal.RTAI: Real-Time Application Interface.Linux Jounral Magazine,2004,4

[32] 张惠娟，翟鸿鸣. 一种固定优先级实时调度算法的可行性测定微机发展，2003，13(9) ：65-67

[33] Wei Zhang, Shaohua Teng,Zhaohui Zhu, Xiufen Fu, Haibin Zhu.An Improved Least-Laxity-First Scheduling Algorithm of Variable Time Slice for Periodic Tasks.Cognitive Informatics, 6th IEEE International Conference on 6-8 Aug. 2007 Page(s):548 – 553

[34] Yu Xin.Yu Shao-hua.Huang Ben-xiong. Zhang Shi-jun.A DMR Fair Algorithm for Realtime Scheduler.Future Generation Communication and Networking, 2008

[35] De la Rocha, F.R.de Oliveira, R.S.A real-time task model based on ideal instant. Emerging Technologies and Factory Automation, 2005. ETFA 2005. 10th IEEE Conference onVolume 1, 19-22 Sept. 2005 Page(s):4 – 1088

[36] 朱响斌，涂时亮. Linux 的实时性能测试. 微电子学与计算机，2004，21(11)，85-88

硕士在读期间发表论文和成果

论文发表情况

[1] 朱春飞，曹伯燕，崔巍. Linux 实时调度机制及改进策略分析. 西安电子科技大学 2008 年研究生学术年会.

附录

1. 系统测试模块

```
#define TICK_PERIOD 800000   //任务task0的周期，单位为ns;
#define TASK_PRIORITY 1 //所有任务的优先级定位1;
#define STACK_SIZE 10000
#define FIFO 0
Static RT TASK rt_task[10] //10个任务控制块，分别对应10个任务
Static void test0(int t)   //任务0的主函数
{
    char start[]="0 began";
    char wait[]="0 waited ";
    static int counter;
    while (1){
        rtf_put(FIFO,start,8);//将"0 began"输出到FIFO
        counter=0;
        do
        {
            counte r++;
        }
        while(counter<10000)
        rtf_put(FIFO,wait,8);//将"0 waited"输出到FIFO
        rt_task_wait_period();// 进入周期性休眠
    }
}
```

2. 系统初始化模块

```
int init_module(void)
{
    RTIME tick_period;
    int i;
    rt_set_periodic modes;
    rt_task_init(&rt_task[0],test0,1,STACK_SIZE,TASK_PRIORITY,1,0);
    rt_task_init(&rt_task[l],test1,1,STACK SIZE,TASK_PRIORITY,1,0);
    ……
    rt_task_init(&rt_task[9],test9,1,STACK_SIZE,TASK_PRIORITY,1,0);
    rtf_create(FIFO,8000);
    for(i=0;i<10;i++){
        tick_period=start_rt_timer(nana2count(TICK_PERIOD+i*1000));
        rt_task_make_periodic(&rt_task[i],rt_get_time()+tick_period,tick_period);
        rtai_set_task_deadline(&rt_task[i],nana2count(TICK_PERIOD+(9-i)*1000))
    }
    rtai_make_schedule_alg(SCHED_RM);
    return 0;
}
```

3. 清除函数

```
void cleanup_module(void)
{
    stop_rt_timer();
    rtf_destory(FIFO);
    for(i=0;i++;i<10)
    rt_task_delete(&rt_task[i]);
```

```
    return ;
}
```

4. 系统打印函数

```
int main(void)
{
    char result[8];
    float sin_value;
    if((fifo=open("/dev/rtf0",O_RDONLY)
    {
        fprintf (stderr,"Error opening/dev/rtf0\n");
        exit(1);
    }
    while(1)
    {
        read(fifo,char,8);
        printf("task%s\n",result);
    }
}
```

5. 任务响应时间测试主模块

```
void
fun(long thread)
{

    int diff = 0;
    int i;
    int average;
    int min_diff = 0;
    int max_diff = 0;
    RTIME t, svt;

    svt = rt_get_cpu_time_ns();
    samp.ovrn = 0;
    while (1) {

        min_diff =   1000000000;
        max_diff = -1000000000;

        average = 0;

        for (i = 0; i < loops; i++) {
            cpu_used[hard_cpu_id()]++;
            expected += period_counts;

            if (!rt_task_wait_period()) {
                if (timer_mode) {
                    diff = (int) ((t = rt_get_cpu_time_ns()) - svt - period);
                    svt = t;
                } else {
                    diff = (int) count2nano(rt_get_time() - expected);
                }
            } else {
                samp.ovrn++;
                diff = 0;
                if (timer_mode) {
                    svt = rt_get_cpu_time_ns();
                }
```

```
            }

            if (diff < min_diff) { min_diff = diff; }
            if (diff > max_diff) { max_diff = diff; }
            average += diff;
#ifdef CONFIG_RTAI_FPU_SUPPORT
            if (use_fpu) {
                dotres = dot(a, b, MAXDIM);
            }
#endif
        }
        samp.min = min_diff;
        samp.max = max_diff;
        samp.index = average / loops;
        rtf_put(DEBUG_FIFO, &samp, sizeof (samp));
    }
    rt_printk("\nDOT PRODUCT RESULT = %lu\n", (unsigned long)dotres);
}
```

6. 中断延时时间测试主模块

```
static void fun(long thread) {

    struct sample { long min, max, avrg, jitters[2]; } samp;
    int diff;
    int skip;
    int average;
    int min_diff;
    int max_diff;
    RTIME svt, t;

    min_diff = 1000000000;
    max_diff = -1000000000;
    while (1) {
        unsigned long flags;
        average = 0;

        svt = rt_get_cpu_time_ns();
        for (skip = 0; skip < NAVRG; skip++) {
            cpu_used[hard_cpu_id()]++;
            expected += period;
            rt_task_wait_period();

            rt_global_save_flags(&flags);
            if (diff < min_diff) {
                min_diff = diff;
            }
            if (diff > max_diff) {
                max_diff = diff;
            }
        average += diff;
        }
        samp.min = min_diff;
        samp.max = max_diff;
        samp.avrg = average/NAVRG;
        samp.jitters[0] = fastjit;
        samp.jitters[1] = slowjit;
        rtf_ovrwr_put(FIFO, &samp, sizeof(samp));
    }
```

```
}
```

7. 任务切换时间测试主模块

```
static void sched_task(long t) {
    int i, k;
    unsigned long msg;

    change = 0;
    t = rdtsc();
    for (i = 0; i < loops; i++) {
        for (k = 0; k < ntasks; k++) {
            rt_task_resume(thread + k);
        }
    }
    t = rdtsc() - t;
    rt_printk("\n\nFOR %d TASKS: ", ntasks);
    rt_printk("TIME %d (ms), SUSP/RES SWITCHES %d, ", (int)llimd(t, 1000,
CPU_FREQ), 2*ntasks*loops);
    rt_printk("SWITCH TIME%s %d (ns)\n", use_fpu ? " (INCLUDING FULL FP
SUPPORT)":"",
            (int)llimd(llimd(t, 1000000000, CPU_FREQ), 1, 2*ntasks*loops));

    change = 1;
    for (k = 0; k < ntasks; k++) {
        rt_task_resume(thread + k);
    }
    t = rdtsc();
    for (i = 0; i < loops; i++) {
        for (k = 0; k < ntasks; k++) {
            rt_sem_signal(&sem);
        }
    }
    t = rdtsc() - t;
    rt_printk("\nFOR %d TASKS: ", ntasks);
    rt_printk("TIME %d (ms), SEM SIG/WAIT SWITCHES %d, ", (int)llimd(t, 1000,
CPU_FREQ), 2*ntasks*loops);
    rt_printk("SWITCH TIME%s %d (ns)\n", use_fpu ? " (INCLUDING FULL FP
SUPPORT)":"",
            (int)llimd(llimd(t, 1000000000, CPU_FREQ), 1, 2*ntasks*loops));

    change = 2;
    for (k = 0; k < ntasks; k++) {
        rt_sem_signal(&sem);
    }
    t = rdtsc();
    for (i = 0; i < loops; i++) {
        for (k = 0; k < ntasks; k++) {
            rt_rpc(thread + k, 0, &msg);
        }
    }
    t = rdtsc() - t;
    rt_printk("\nFOR %d TASKS: ", ntasks);
    rt_printk("TIME %d (ms), RPC/RCV-RET SWITCHES %d, ", (int)llimd(t, 1000,
CPU_FREQ), 2*ntasks*loops);
    rt_printk("SWITCH TIME%s %d (ns)\n\n", use_fpu ? " (INCLUDING FULL FP
SUPPORT)":"",
            (int)llimd(llimd(t, 1000000000, CPU_FREQ), 1, 2*ntasks*loops));
}
```