华中科技大学

硕士学位论文

嵌入式实时操作系统ARTs-OS的EDF调度算法改进

姓名：陈磊

申请学位级别：硕士

专业：软件工程

指导教师：刘云生

2011-01-13

摘 要

随着人们在生产、生活中对实时处理需求的不断增多，嵌入式实时操作系统（Embedded Real-Time Operating System,简称 ERTOS）的应用越来越广泛。实时调度算法对嵌入式实时操作系统的实时性有重要影响，是提高系统实时性的关键技术。

ARTs-OS 系统是实验室自主研发的嵌入式实时操作系统。系统采用微内核结构，内核体积小巧，可以动态加载模块。任务管理模块使用多进程多线程模式，进程之间有内存保护，使得系统性能稳定。调度模块使用了面向对象的思想，通过实现调度器对象，可以方便地在不影响系统其它模块的情况下修改调度方式。

调度的实质是 CPU 资源的分配和利用。EDF 算法作为一种最优的动态优先级调度算法，运行开销小、处理器利用率高，但是调度开销大，优先级决定机制过于单一不能体现任务的重要性，在系统过载的情况下会产生“多米诺效应”造成大多数任务的实时要求得不到满足。通过引入一个表示任务重要程度的因子与任务的截止时间共同决定任务的优先级，可以更充分地体现应用要求。系统过载时，在 EDF 算法的可调度范围内选取重要的任务作为调度对象优先调度，可以避免“多米诺效应”带来的影响。

将改进后的 EDF 算法在 ARTs-OS 系统上实现，调度不同负载下的任务集。从实验结果可以看出系统能够对任务进行有效调度，在系统过载情况下也能保证重要任务的实时要求先得到满足。

关键词：嵌入式实时操作系统 实时调度算法 EDF 调度算法

Abstract

With the constant growing demand for real-time processing in people's production and life, Embedded Real-Time Operating System (ERTOS) is used more and more widely. Real-Time Scheduling algorithms have an important effect on the real-time performance of ERTOS, is the key to improve the real-time performance.

ARTs-OS is an embedded real-time operating system which is designed and developed by ourselves laboratory. The system uses micro-kernel structure, the size of the kernel is small and user can load modules dynamically. The task management module uses process/thread model, has memory protection among processes which makes the system more stable. Object Oriented Programming is introduced into the designing of scheduling module. User can develop a new scheduling object to change the scheduling way instead of changing most modules of the system.

The essence of scheduling is the allocation and use of processor resources. As an optimal dynamic priority scheduling algorithm, EDF algorithm has low running cost and high processor utilization. But the scheduling cost of EDF algorithm is large, and the priority decision mechanism is too simple can't reflect the importance of the tasks. When the system overloads, EDF algorithm will have a "domino effect", causing most of the tasks miss their deadline. Using a factor which reflects the importance of task and deadline to decide the priority of task, the application requirement can be better met. Selecting important tasks to scheduling can avoid the "domino effect".

Apply the improved EDF algorithm to Arts-OS, schedule tasks under different system environment. The experimental results show that when the system is overloading the improved EDF algorithm can assure the requirement of important tasks firstly.

Key words: ERTOS real-time scheduling algorithm EDF scheduling algorithm

独创性声明

本人声明所呈交的学位论文是我个人在导师指导下进行的研究工作及取得的研究成果。尽我所知，除文中已经标明引用的内容外，本论文不包含任何其他个人或集体已经发表或撰写过的研究成果。对本文的研究做出贡献的个人和集体，均已在文中以明确方式标明。本人完全意识到本声明的法律结果由本人承担。

学位论文作者签名：

日期：　　年　　月　　日

学位论文版权使用授权书

本学位论文作者完全了解学校有关保留、使用学位论文的规定，即：学校有权保留并向国家有关部门或机构送交论文的复印件和电子版，允许论文被查阅和借阅。本人授权华中科技大学可以将本学位论文的全部或部分内容编入有关数据库进行检索，可以采用影印、缩印或扫描等复制手段保存和汇编本学位论文。

本论文属于　保密□，　在_____年解密后适用本授权书。

　　不保密□。

（请在以上方框内打“√”）

学位论文作者签名：　　指导教师签名：

日期：　　年　月　日　　日期：　　年　月　日

1 绪论

随着电子信息技术的不断发展，越来越多的应用对系统实时性能提出了要求。嵌入式实时操作系统是一种能够提供实时处理功能的操作系统，已经被广泛应用于工业控制、军事设备、航空航天和通信传输等领域中。

嵌入式操作系统处于系统硬件和软件之间，能在有限的资源下为用户提供内存管理，多任务管理，中断处理和时钟等服务。嵌入式实时操作系统主要包括硬件抽象层、系统内核和用户接口三部分。硬件抽象层与具体的硬件相关，主要包括一些底层驱动，用于驱动和控制硬件。系统内核是系统的核心，由中断、内存和任务等模块组成。用户接口是系统给用户提供服务的通道，用户可以使用接口构造各种复杂的应用程序。嵌入式实时操作系统的结构可以分为单体内核结构和微内核结构，单体结构的内核提供所有的系统服务，但是稳定性差，体积大；微内核结构的内核只提供最基本的系统服务，体积小，稳定性好，但是由于消息传递需要额外的开销，性能不如单体结构[1]。嵌入式实时操作系统一般体积小、可以根据实际需要对软件进行裁剪。

作为一种实时系统，嵌入式实时操作系统对时间具有敏感性。在 POSIX1003.b 标准中，实时系统是指能够在规定的响应时间内提供所需要级别服务的系统。因此嵌入式实时操作系统的性能不仅仅依赖于系统计算结果的正确性，还取决于系统响应和处理事务的时间[2]。根据系统任务对时间的要求紧迫程度的不同，系统可以分为强实时操作系统和弱实时操作系统。

（1）强实时操作系统(也叫硬实时操作系统)中的任务必须在规定的时间或者确定的时间内完成，否则将会对系统和应用程序产生严重的后果。这种系统往往应用于对时间有严格要求的领域，例如军事领域和航天领域，如果不能及时准确地得到结果，可能造成不能精确命中目标和偏离航道。

（2）软实时操作系统对时间的要求则宽松一些，如果有任务未在规定的时限内完成，只是造成一定的延迟，不会对系统和应用程序造成太大的损失，对于用户而

言是可以接收的，如视频会议中的短暂延迟并不会对用户造成太大影响。

此外嵌入式实时操作系统具有可预测性和可靠性。系统分配资源的时间和执行基本功能的时间应该有个最大值，以保证任务需要等待的时间是确定的。当出现错误时，系统应该提供一定的容错机制保障系统能够正常运行。

1.1 嵌入式实时操作系统介绍

嵌入式实时操作系统自上世纪七十年代产生以来，进过多年发展技术日趋成熟。国内外已经涌现了不少优秀的嵌入式实时操作系统，这些系统在人类的生产、生活和科研中都得到了广泛地应用。比较常用的嵌入式实时操作系统有 uC/OS-II、RTLinux、Vxworks、WindowCE、uClinux 等。

uC/OS-II 是一种源码开放、微内核结构的嵌入式实时操作系统。其前身是 uC/OS, 于 1992 年由美国嵌入式系统专家 Jean J.Labrosse 在《嵌入式系统编程中》上发布。uC/OS-II 体积小巧，不支持 MMU，不支持同优先级调度，但是支持内核的完全抢占；uC/OS-II 代码多用 C 编写，少量与硬件相关的使用汇编；uC/OS-II 执行效率高，具有良好的实时性、可移植性、可拓展性，已被成功移植到 40 多种处理器上[3]。

VxWorks 由美国 Wind River System 公司在 1983 年设计和开发，采用微内核结构设计。VxWorks 拥有快速灵活的 I/O 接口和丰富的任务通信方式，支持多种文件系统和标准 TCP/IP 协议，有任务切换时间小、中断延迟时间短、网络流量大等特点；Vxworks 拥有友好的用户交互界面和丰富的扩展组件，加上良好的可靠性和实时性，被广泛用在航空航天、军事、通信等实时性要求比较严格的领域，在国内外嵌入式系统市场上占有很大的份额[4]。

WindowCE 是微软公司推出的一种高效、可升级的嵌入式实时操作系统。C 表示消费(Consumer)、袖珍(Compact)、伴侣(Companion)和通信能力(Connectivit)，E 表示电子产品(Electronics)，是微软专门为嵌入式设备开发的平台[5]。WindowCE 支持多线程，多任务处理，采用基于优先级的强占式调度。它的用户界面继承了传统的 Windows 的图形界面，编程接口也类似。此外 WindowCE 采用模块的设计方案，可以根据需要搭建出各种平台。

RTLinux、uClinux 嵌入式操作系统。Linux 操作系统是一种源码开发的软件，功能较为完善，稳定性、可靠性也得到了实践检验。但是由于其体积庞大，调度时以提高系统平均吞吐率为目的，本质上并非实时操作系统，所以并不适合直接做嵌入式实时操作系统。但是可以对 Linux 系统进行剪裁和改进，使得其符合嵌入式实时系统的要求，RTLinux 和 uClinux 就是以 Linux 系统为基础开发的嵌入式实时操作系统。RTLinux 是由美国新墨西哥州的 fsmlabs(finite state machine labs,有限状态机实验室)公司开发，它在硬件层之上实现了一个实时内核，并把 Linux 内核和 Linux 中的任务做为非实时任务与实时任务一起调度；uClinux 是有 Lineo 公司开发的高度优化和精简的嵌入式 linux 系统,它代码精炼、体积小，主要用于没有存储管理的处理器，符合 GNU/GPL 公约，拥有丰富的 API，专用性很强，份额占有率居全球嵌入式 Linux 市场第二[6]。

1.2 嵌入式实时操作系统性能影响因素

实时性是嵌入式实时操作系统的重要特点，也是衡量其好坏的重要因素。系统实时性常用进程分派延迟法、三维表示法和 Rhealstond 法来测量[7]。要想实现高效嵌入式实时操作系统必须考虑以下几个关键技术：中断处理、内存管理、时钟机制和任务管理等。

中断是系统响应外部事件的一种方式。根据中断机制的不同系统可以分为可抢占内核和不可抢占内核[8]。可抢占内核是指中断过程中如果有高优先级的任务就绪，中断结束后高优先级的任务可以抢占当前任务得到运行；不可抢占内核中，就绪的高优先级任务必须等待当前任务执行完后才能调度。为防止新中断的丢失，系统关中断的时间应该尽量小，不重要的操作可以稍后执行。嵌入式实时操作系统一般是可抢占内核。

嵌入式实时操作系统的内存管理与普通系统不一样，要求内存分配具有快速性、可靠性和高效性[9]。系统的实时处理要求尽可能快并且准确地进行内存分配。由于内存资源有限，系统往往希望能高效、合理地利用内存。常用的内存分配策略有静态内存分配和动态内存分配，静态内存分配一般内存大小固定，可事先确定，速度快，

但使用率不高，适用于有硬实时要求的系统；动态内存分配灵活，利用率高，但是效率不及静态内存分配，适用于软实时操作系统。

时钟是系统的脉搏，为系统提供计时和定时功能。时钟粒度是操作系统的最小计时单位，直接决定了时钟精度。细粒度的时钟可以增加系统抢占点和系统调度的时机、提高系统对外界的响应速度。因此细化时钟粒度和提高时钟精度有利于提高系统的实时性。细化时钟粒度可简单地通过提高时钟频率来实现，但是过密的时钟中断减小了系统处理其它任务的时间，系统性能反而下降[10]。提高时钟精度可以通过TIMER机制、RFRTOS TIMER机制和RTOS KERNEL TIMER机制实现，其中RTOS KERNEL TIMER 机制使用两个时钟，一个用于正常周期计时的普通时钟和一个在需要时设置的高精度时钟效果较好[11]。

任务管理重点在于进程模型的选择和调度机制的使用。进程模型一般有单进程模型、多进程模型、单进程多线程模型和多进程多线程模型[12,13]。调度机制包括调度方式和调度算法。根据调度时机的不同，调度方式可以分为不可抢占式调度和可抢占式调度，不可抢占调度中系统仅在控制从内核态返回用户态时启动调度器，而可抢占调度启动调度器的条件相对宽松些[14]。嵌入式实时操作系统一般采用可抢占调度。此外，调度算法也在很大程度上决定了系统的实时性。本文将以实时调度算法为研究对象，讨论如何通过实时调度算法提高系统的实时性能。

1.3 实时调度算法研究现状

实时调度算法决定了系统内任务执行的先后顺序，好的调度算法可以保证任务在其时限内完成，提高系统的实时性能。因此实时调度算法一直是国内外研究的热点领域。

实时调度算法调度的对象是周期任务和非周期任务。周期任务是每隔一段固定时间到来的任务，因为周期任务到来的时间有规律，其调度算法已经比较成熟，有RM 算法、EDF 算法和 LLF 算法等；非周期任务的到来有随机性，常与周期任务混合调度，已有的解决方法包括后台运行法、保留带宽法和时间挪用法[15]。其中后台运行法是将非周期任务放置后台，等待处理器空闲时再运行；带宽保留法是系统为

非周期任务准备一个周期任务的执行时间，在这个时间段内执行非周期任务；时间挪用法是把周期任务中可挪用的空余时间用于执行非周期任务。

根据调度依据对象的不同实时调度算法可以分为时间驱动的调度、优先级驱动的调度和处理器时间比例共享的调度[16]。其中优先级驱动调度算法应用最为广泛，又分为静态优先级调度和动态优先级调度。静态优先级调度中任务的优先级可以预测、额外开销小、稳定性好，但是灵活性差、不能适应不可预测环境，常见的静态优先级调度算法有 RM 算法；动态优先级算法灵活性好、但系统开销大、而且只能决定当前任务的顺序不考虑将要到达的任务，常见的动态优先级算法有 EDF 算法和 LLF 算法[17]。RM 算法已经被证明是最优的静态优先级调度算法，EDF 算法和 LLF 算法也被证明是最优的动态优先级调度算法[18].实际应用中静态优先级调度算法由于简单方便应用较多；而动态优先级算法由于系统开销大等一些不足应用较少，多以可选择的调度方式提供给用户。

根据算法能否自动依据系统状态的变化而改变，实时调度算法又可以分自适应调度和非自适应调度，自适应调度可分为基于 CPU 资源控制的方法、基于准入控制的方法和基于 QoS 控制的方法[19].

根据处理器运行平台的不同，实时调度算法可分为单处理器调度、多处理器调度和分布式调度。传统的调度算法多以单处理器为主。近年来由于多处理器的出现和使用，多处理器调度研究也越发重要。多处理器调度比单处理器调度复杂，因为算法不仅需要决定任务调度的顺序，还要决定任务分配到那个处理器执行，有全局和划分两种调度策略，经典算法有 Pfair 算法[20,21]。分布式调度算法主要用于多媒体领域是实时调度算法的另一热点领域。

目前实时调度算法的研究方向主要集中在对传统实时调度算法的改进、对周期任务和非周期任务的混合调度以及多处理器、多媒体系统的实时调度等方面。

1.4 论文的研究内容和组织结构

研究实时调度算法对提高的实时性有非常重要的作用。目前对于单处理器调度，已有的动态优先级算法理论已经比较成熟，但是在实际应用中局限较大。本文将选

取一种动态实时调度算法 EDF 算法做为研究对象，从应用的角度对其进行具体分析和改进。并将改进后 EDF 调度算法应用到现有的嵌入式实时操作系统 ARTs-OS 中，讨论其在实际应用中的调度效果。

本文第二章将对 ARTs-OS 系统的实时调度机制进行详细介绍。

第三章将对 EDF 实时调度算法进行具体分析，找出其优点和不足。并从应用的角度提出改进方案，给出具体算法描述。

第四章详细描述在 ARTs-OS 中实现改进后 EDF 调度算法的具体步骤和方法。

第五章对实现在 ARTs-OS 系统上的改进后 EDF 算法进行测试，分析其调度效果。

第六章对本文的研究工作做总结，提出本文的不足和有待研究的地方。

2 ARTs-Os 实时调度机制

ARTs-OS 是一种采用微内核结构的嵌入式实时操作系统，由作者所在实验室自主设计和开发，可以广泛应用于国防武器、工业生产过程控制、实时多媒体信息服务、通信设备、交通控制和信息化家电等领域。由于采用了现代微内核设计思想和模块结构，ARTs-OS 内核的体积小巧。ARTs-OS 支持内核动态化机制，用户可以动态地加载和卸载内核模块，进而灵活、方便地配置系统。此外 ARTs-OS 符合实时 POSIX 标准、支持多种 CPU、采用多进程多线程模型、支持硬实时性能、具有确定性和可预测性，是一种高性能的嵌入式实时操作系统。

2.1 ARTs-OS 系统结构

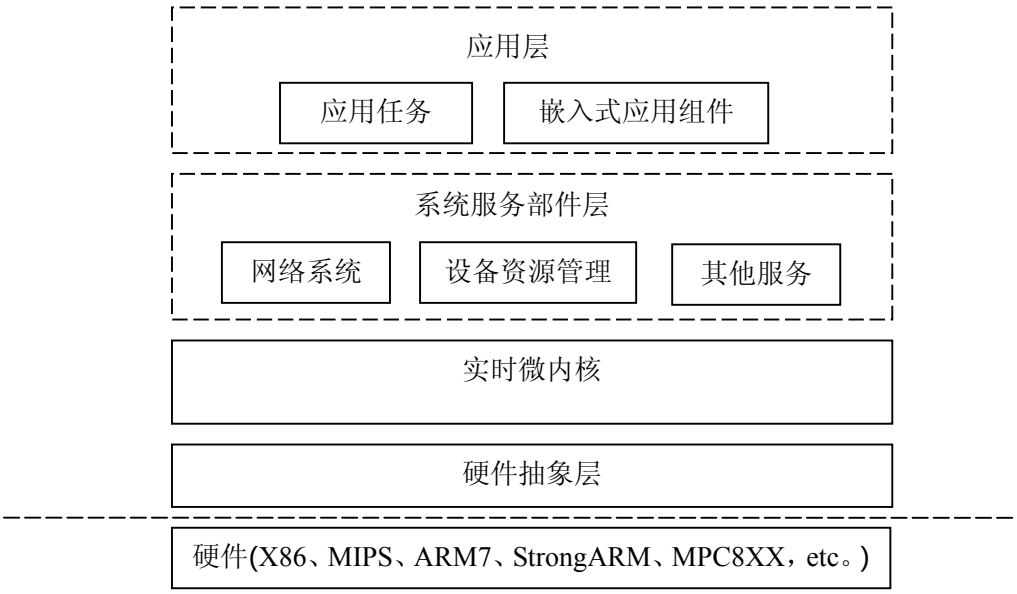

图 2-1　ARTs-OS 系统结构

ARTs-OS 整个系统结构如图 2-1 所示，可以分为：硬件抽象层、实时微内核、系统服务部件层和应用层。

硬件抽象层介于硬件电路和操作系统内核之间，位于操作系统的最底层，主要包括一些基本设备的驱动程序和中断管理部分所有与平台相关的代码等。硬件抽象层是系统硬件和系统软件的接口，用于向下屏蔽硬件和向上为系统提供虚拟服务。

当系统需要移植时只需要修改与具体平台相关的硬件抽象层，无需修改整个内核，减少了开发人员的工作。因此 ARTs-OS 具有较好的系统移植性，目前已经成功移植到 X86、ARM7 和 ARM9 等处理器上。

ARTs-OS 的内核是微内核，体积非常小，最小可达几十 KB。另外由于采用了模块化的结构，用户可以根据需要动态的加载和卸载内核模块，使得内核易于裁剪和扩充。由于使用的是微内核设计方法，ARTs-OS 内核只提供最基本的内核服务，包括：内存管理、任务管理、中断和时钟管理以及进程间通信等。内存管理包括虚拟内存块（VM）和用户堆（Userheap）两个模块。ARTs-OS 系统的虚拟内存管理与传统意义上的虚拟内存不一样，它是把进程的内存地址空间做为一个整体来操作和管理，以达到高效合理地利用进程地址空间和保护各个进行地址空间的目的。用户堆管理模块采用静态分配和动态分配相结合的方法，反应速度快、效率高、系统开销小、适用于少量多次的内存请求。ARTs-OS 的中断管理主要处理与 IO 设备相关的中断，分为核外中断和核内中断，并提供多层中断嵌套和核外线程中断挂接的功能，部分实现内核抢占机制。时钟管理主要提供系统计时和定时的功能。ARTs-OS 内核的另一大亮点就是提供了丰富的 IPC 用于实时线程的通信和同步，包括：信号机制、信号量机制、内存共享和消息队列。ARTs-OS 的信号量使用优先级继承方法解决了由于系统抢占引起的优先级颠倒问题。另外 ARTs-OS 的消息机制通过使用内存映射减少了消息在内核拷贝的次数。这些技术都从不同方面提高和完善了 ARTs-OS 系统的实时性能。

系统服务部件层主要是指在核外用户空间以核外进程形式运行的系统服务，比如:嵌入式 TCP/IP 协议栈和设备资源管理器。在传统结构的操作系统中，这些服务往往是作为驱动程序放置在内核。但是在微内核结构的操作系统中，由于内核仅仅提供基本的系统服务，其余的系统服务就必须以核外进程的方式运行。此时内核就作为一个服务器向作为用户的核外服务线程提供服务。这种结构还有一个优点，就是可以根据用户需要动态地加载和卸载服务程序，减小不必要的系统开销。

应用层位于操作系统的上层，是给用户使用的。ARTs-OS 在这一层为用户提供了丰富的 API，用户可以直接使用这些接口来编写应用程序。同时 ARTs-OS 也支持

一些嵌入式应用组件供用户拓展使用，如嵌入式 JAVA 和嵌入式 GUI 等。

2.2 ARTs-OS 进程模型

ARTs-OS 支持多任务处理，其任务管理采用多进程多线程模式。这种模式是指把系统中的每个任务做为一个进程，进程中可以有多个执行流但是至少包含一个执行流。每个执行流是一个线程，系统调度线程来运行。多进程多线程模式下，操作系统有多个进程，系统为每个进程分配固定的内存空间，进程之前有内存保护，不能随意互访。每个进程可以有多个线程，一定有一个主线程，其余线程由此主线程派生。进行内的线程可以并发执行，运行效率高。

多进程多线程模式相对于其它进程模式具有安全性、稳定性好，执行效率高等特点。对于单进程模型和单进程多线程模型，系统和用户共享同一地址空间，系统地址缺乏保护，用户可以很容易地访问到系统，系统的稳定性和安全性得不到有效保证，但是由于共享地址空间，系统执行效率较高；对于多进程模型，进程有自己的地址空间，系统安全性好，不过由于需要管理每个进程的资源，执行效率不高。多进程多线程模式则兼顾了上述两种模式的优点。

ARTs-OS 的进程管理提供的服务包括：进程的创建、执行、退出和状态的查询。进程，前而言之就是一个运行的程序。程序运行时需要空间和资源。系统创建进程时会为每个进程分配所需的地址空间和资源。在进程结束时会自动收回所分配的地址空间和资源。进程之间不能随意访问，必须通过进程间的通信机制比如 IPC 等。每个进程必须有一个线程，否则不能执行。ARTs-OS 的一些系统服务就以核外用户级进程的方式执行。

ARTs-OS 的线程管理模块主要用于：支持线程的创建、阻塞、唤醒和退出；支持线程相关信息的查询和状态的维护。线程是任务的执行流，任何一个线程都属于一个进程，同属于一个进程的线程共享同一地址空间和系统资源，因而没有地址空间的保护。线程所占用的资源比进程少，控制也比进行进程简单，所以线程切换的开销要比进程切换小，系统把线程作为调度的基本单位。线程在系统运行时有就绪（ready）、运行（running）、阻塞（blocked）和终止（Dead）状态。各种状态的转换

如图 2-2 所示。当一个线程被创建时或者一个本处于阻塞状态的线程由于获得资源或者消息和事件的到来而被唤醒时处于就绪状态。当一个线程被处理器调度执行时处于运行状态。当一个线程需要等待资源、消息和事件的到来而暂时挂起时，处于阻塞状态。当一个线程由于某种原因停止运行，不参与系统调度但是仍然保留其线程控制块时处于终止状态。

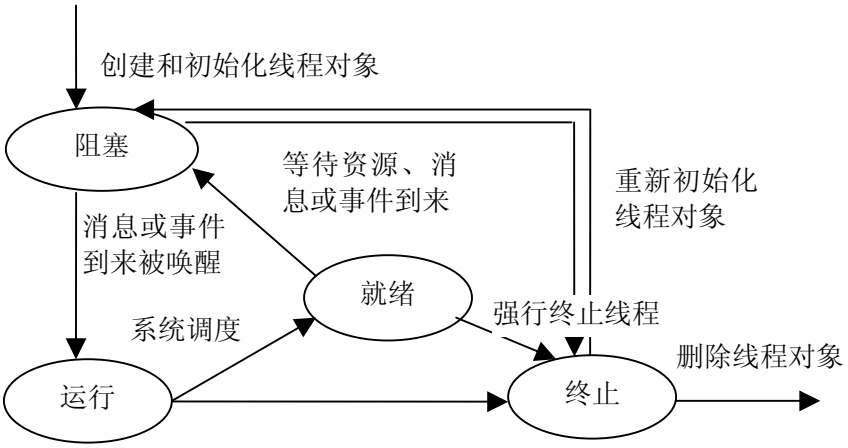

图 2-2 ARTs-OS 线程状态转换

2.3 ARTs-OS 调度方式

ARTs-OS 采用的是基于优先级的可抢占调度。此外，系统提供的高精度的计时和定时功能也极大的支持了系统的实时调度。

2.3.1 基于优先级调度

ARTs-OS 以线程为基本单位进行调度，采用了基于优先级的调度策略。线程在创建时由系统根据优先级分派方法分配一个优先级，在运行中可以动态地设置此优先级，并在下一次调度中启用新设置的优先级。系统调度时调度器总会从就绪线程中选取优先级最高的线程来运行。

ARTs-OS 中提供的优先级范围在 0~95。优先级值越小，优先级越高。为标识应用中的不同时限要求的线程，系统将优先级分为三类 。第一类优先级值在 0~31，表示硬实时任务；第二类优先级值在 32~63，表示软实时任务；第三类值在 64~95，表示非实时任务。

目前在基于优先级调度的基础上，ARTs-OS 已经结合 FIFO 和 RR 算法实现了基

于优先级的 FIFO 和 RR 调度方式。这两种调度的区别在于对于优先级相同的线程，FIFO 选取本优先级中最先到来的任务运行，而 RR 则让本优先级的线程公平地使用 CPU 资源。

2.3.2 可抢占调度

在系统运行过程中，系统启动调度器重新调度线程，使得高优先级任务有机会抢占低优先级任务的时刻称之为调度的可抢占点。对于不可抢占调度的系统，系统启动调度器的时机较少，高优先级的任务就绪后必须等待正在运行的线程执行完毕后才能参与调度。对于可抢占调度的系统，系统启动调度器的时机比较多，高优先级的任务在就绪后可以在随后的抢占点中抢占低优先级运行，不必等到低优先级任务执行完毕。

ARTs-OS 采用的是可抢占调度方式。系统启动调度器的原因可以分为主动和被动俩方面。主动方面主要是指线程主动放弃处理器，比如线程执行完毕、线程需要等待资源而阻塞自己等。被动方面主要是指由于中断等到来而引起的抢占，这里的中断包括硬件中断和软件中断。比如一个定时器到时将唤醒一个阻塞的线程，如果此线程优先级高，那么它将抢占当前线程运行。同样如果一个消息的到来会使线程从阻塞状态转为就绪状态，参与调度。无论是主动还是被动，ARTs-OS 都是通过在中断处理的尾部实现抢占的。

2.3.3 高精度时钟支持

ARTs-OS 提供的时钟服务包括系统计时和定时功能。计时功能包括记录系统时间和实现时间片轮转的调度等，用户可以通过应用接口查询和配置系统时间。ARTs-OS 的定时器提供了高精度的定时功能，最小定时时间可为 4 微秒。用户可以通过设置定时器让系统到时后执行指定程序。

ARTs-OS 的时钟服务是通过使用 PIT(Programmable Interval Timer 可编程间隔定时器)来完成的。向 PIT 设置预定的时间，超时后 PIT 会产生中断。考虑到不必要的定时器中断会影响系统的性能。ARTs-OS 在设计时钟服务时使用了两个定时器，一个低精度的定时器和一个高精度的定时器。低精度定时器用于控制系统的节奏，采

用自动装载模式产生周期中断，定时长度固定为 10 毫秒，系统每过 10 毫秒产生一次时钟中断。系统中有一个变量 jiffies，用于记录系统自开启一来的时钟滴答数，即每当低精度的时钟中断时，jiffies 的值就会加 1。高精度的时钟用于实现用户的定时功能。用户设置的定时要求按照图 2-3 所示排列成一个链表。

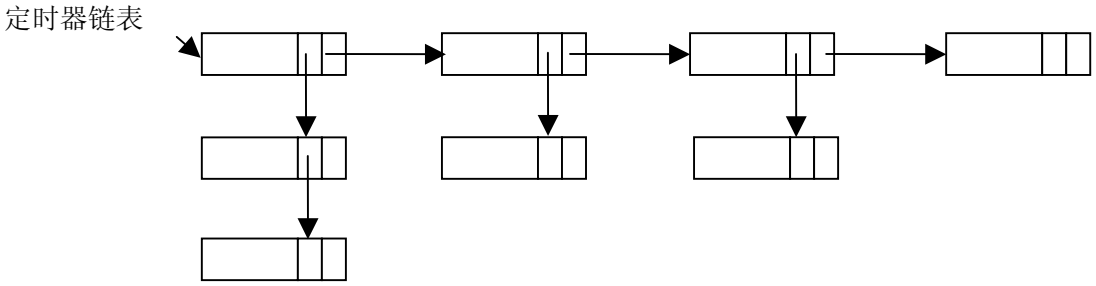

图 2-3 ARTs-OS 定时器链表

横向链表中定时器定时值为链表中其左边的所有定时器的定时值之和。当有新的定时器到来时，需要从左到右的搜索定时器横向链表，比较新定时器的时间与横向链表的时间。如果新定时器时间大，则将其减去所比较的定时器时间后，继续向后比较。如果新定时器的时间小，则将新定时器放置到所比较的定时器前面。纵向链表中的定时器定时值与纵向链表中的第一个定时器相同。每次低精度的时钟产生中断时，系统将搜索定时器链表定时时间最小的第一个项。如果定时器的值大于 10 毫秒，将其时间减去 10 毫秒。如果该定时器的值小于 10 毫秒，就将其设置为高精度的定时器，当高精度的定时器超时产生中断时就会触发该定时器去执行用户操作。ARTs-OS 中高精度定时器与低精度定时器相结合的方法即有效地利用系统中断又提高了计时的精确程度。

2.4 ARTs-OS 调度器

线程是 ARTs-OS 的最小执行单位，是 ARTs-OS 的调度对象。ARTs-OS 的线程调度需要从就绪线程中选取下一个运行线程，如果新选取的线程与旧线程不是同一个线程则需要进行线程切换，保存旧线程的上下文信息，装入新线程的上下文信息。其中选取线程是通过线程调度器实现。

ARTs-OS 的线程调度器采用了面向对象的设计方式，是一种可以动态配置的调

度器。调度器把整个调度过程的实现抽象成一个类，如图 2-4 所示。

图 2-4 ARTs-OS 调度器

调度器把系统里的线程队列作为操作的数据对象，把调度函数接口作为具体的操作方式。调度器里的函数接口都是调度时的通用操作，包括：线程调度初始化、将线程插入优先级队列中、将线程从优先级队列中移出、将线程设置为就绪状态，将线程设置为阻塞状态和根据具体策略执行调度等。其中执行那个调度的函数是整个调度器的核心，是系统调度方式的具体实现。其它的函数用于根据线程的状态转换来维护线程队列

因此要想在 ARTs-OS 上实现新的调度方式，只需要实现一个新的调度器对象就可以了。在不同的实时环境中，用户对优先级的分派会采用不同的方法，此时用户可以根据自己的需要自行编写线程调度器。采用这种方法，可以很容易地实现和修改线程调度器，减少调度器变化对系统其它部分的影响。

2.5 小结

本章对 ARTs-OS 的调度机制进行了详细介绍。ARTs-OS 采用采用微内核结构，可动态加载和配置内核。ARTs-OS 的进程模型采用了多进程多线程模型，进程用于资源分配，线程用于任务执行，是系统调度的基本单位。ARTs-OS 的调度器采用了面向对象的设计方式，用户可以在不影响其它模块的情况下改变系统的调度方式。ARTs-OS 的实时调度采用了基于优先级的可抢占调度，系统为此提供了高精度的时钟支持。

3 EDF 算法分析与改进

作为一种最优的动态实时调度算法，EDF 算法的适应性好、比较灵活、CPU 使用率高，但是 EDF 算法系统开销比较大，不能处理系统过载。本章将对 EDF 算法进行详细分析，然后提出一些改进措施。

3.1 实时调度算法概述

调度的实质是把 CPU 作为一种资源按照一定的策略和算法分配给就绪任务。实时系统的调度与普通系统不同。普通系统多地注重系统的整体性能和资源利用，如系统吞吐量和平均响应时间。实时系统则把保证任务能在有效的时限内完成放在首位考虑。

3.1.1 实时调度概念

实时系统在调度时，尽可能多地满足每个任务时限要求，在满足任务时限的情况下才最求最小响应时间。调度的对象根据到来时间是否有规律可以分为周期任务和非周期任务。

（1）周期任务(Periodic tasks)是由一系列有限或者无限的执行请求（子任务）组成，这些请求的到来是连续有规律的[22]。每两个相邻的请求到来之间相隔一个固定时间，这段间隔时间通常为一个常数值，称为周期 T。假设一个周期任务的某一个请求在 t 时刻到达，则它的下一个请求将会在 t+T 时刻到达。周期任务的请求开始的时刻成为该请求的到达时刻，周期任务的请求就绪的时刻成为该请求的释放时刻。

（2）非周期任务(Aperiodic tasks)的请求到来是无规律的、零散的。系统无法预测非周期任务的到来时刻，如系统无法知道用户在什么时候会按下一个按钮。非周期任务中如果每两个任务之间的间隔之间有个最小值，叫做偶发任务(Sporadic tasks), 这个最小值称作偶发任务的周期 T。如果偶发任务的某一个请求在 t 时刻到达，则其下个请求将会在 t+T 时刻或者 t+T 之后的任意时刻到达，偶发任务往往可以转化成周期任务进行调度[23]。

任务的时间特征可以用任务的周期、到达时间、释放时间、截止时间、执行时间和响应时间等来具体描述[24]。

周期(Period)用于表示任务到来的频率。对于周期任务来说，周期等于相邻两个任务到来的时间间隔，对于偶发任务周期是任务到来间隔中的最小值。

到达时间是指任务到达系统的时刻。释放时间是指任务获得了出除处理器以外的所有资源，转为就绪态，放置到调度队列中的时刻。到达时间和释放时间之间的间隔称为释放抖动[25]。为简化讨论，释放抖动(Release jitter)在实时调度算法的模型中通常假设为零。

截止时间(Deadline)分为相对截止时间和绝对截止时间。相对截止时间(Relative deadline)是指从任务的请求到达开始到该请求必须完成的时间段长度。绝对截止时间(Absolute deadline)是指任务的请求在该时刻之前必须完成。相对截止时间是一个时间段，绝对截止时间是一个时间点。绝对截止时间等于任务请求的到达时刻加上任务的相对截止时间。根据时限要求的严格程度，实时任务又可以分为硬实时任务(Firm task)和软实时任务(Soft task)。在时限之后完成会产生严重后果的任务称为硬实时任务。超过时限完成仍然可以接收的任务称为软实时任务[26]。

执行时间(Execution time)是指为完成任务处理器花费在执行此任务的时间，通常考虑的是任务的最坏执行时间(Worst Case Execution Time)，也就是在最坏情况下，请求需要执行的最长时间。

响应时间(Respone time)是指从任务释放开始到任务执行完毕的这段时间[27]，响应时间表明了任务得到系统服务的程度。响应时间越短，任务得到服务的时间越早；响应时间越长，任务需要等待的时间就越长。

在进行调度分析时，每个任务的处理器需求和整个系统的状态都必须考虑，可用 CPU 使用率和系统负载来作为参考。任务的 CPU 利用率(也称为任务负载，task utilization)等于执行时间与周期之比[28]，表示 CPU 花费在执行该任务的时间占 CPU 总时间的比例。任务集的 CPU 利用率(也称为任务集负载，task set utilization)等于任务集里所有任务的 CPU 利用率之和，即 $U=\sum_{i=1}^{n}\frac{C_i}{T_i}$。系统负载是指系统内所有任务

对处理器需求和系统所能提供的最大服务之间的关系。当所有任务对系统资源的需求在系统可提供的范围内时，系统处于轻载状态，此时任务集的 CPU 使用率小于等于 1；如果任务的需求超过了系统所能提供的服务范围，系统处于过载，此时任务集的 CPU 使用率大于 1。在调度过程中，如果一个任务集中所有的任务都能在时限内完成，称该任务集是可调度的；如果有一个任务超过了时限，没有按时完成，则该任务集是不可调度的[29]。

3.1.2 常见的实时调度策略

常见的实时调度策略有静态表驱动的调度、优先级驱动的调度和比例共享的调度[30]。

静态表驱动（也叫时间驱动）的调度是一种静态的离线调度方式。任务在调度前都有明确的时间需求，系统在调度前根据每个任务的需求确定调度顺序并固定在调度表中，调度时就按照调度表中的顺序对任务进行调度。这种调度策略有较好的预测性，但是缺乏灵活性。如果任务稍有变动整个调度就必须重新安排，所以静态表驱动的调度比较适合于有明确输入输出的控制应用中。

优先级驱动的调度是一种最常用的调度策略。在系统中，每一个需要调度的任务都会分配到一个优先级用于参与系统调度。系统从高优先级到低优先级的顺序调度任务。这种调度方式简单明了，易于实现。根据优先级在任务运行中是否变化又可以分为静态优先级调度和动态优先级调度。静态优先级调度是指任务开始时就由用户或者系统分配一个静态的优先级，用它参与系统调度，并且在任务存在过程中，该优先级保持不变。静态优先级调度算法实现起来简单、方便动态优先级调度是指任务在开始时系统并不马上给予优先级，而是在系统运行中或者调度时，由一个或者多个参数共同决定给出优先级的调度方式。动态优先级调度相对与静态优先级调度要复杂一些，但是更灵活，更能适应环境的变化。优先级驱动的调度算法会产生优先级倒置现象。优先级倒置是指高优先级的任务由于需要等待进入临界区、共享资源而阻塞，低优先级的任务将先于高优先级任务执行的现象。现在已经有许多解决优先级倒置的方法，比如不允许任何任务在临界区中执行时被抢占；优先级继承协议和优先级上限协议等。

基于比例共享的调度，其思想是把 CPU 的使用做为一个资源按照各个任务所占的比重分配给任务使用。任务所占的比重越大，得到使用 CPU 的时间越长；所占比重越小，可以使用的 CPU 时间也越小。任务所占的比重可以用多种方式表现和计量，比如通过改变任务在就绪队列的位置，增加任务被调度的次数；通过增加任务分配到的时间片等等。在软实时多媒体等环境中，这种策略使用的比较广泛。常见的基于处理器共享的调度算法有彩票调度、轮转调度和公平调度等。

上述三种实时调度策略中，静态表驱动的方式一般用于有固定输入输出的工业控制，比例共享的调度多用于多媒体软实时调度，而优先级驱动的方式因为简单、高效，在实际中得到了广泛地应用。下面介绍几种常见的优先级驱动的实时调度算法：RM 算法、EDF 算法和 LLF 算法。

3.1.3 RM 算法

RM(Rate monotonic，单调速率算法)是一种静态优先级的调度算法，由 Liu 和 Layland 于 1973 年提出[18]。算法认为任务到达的频率越高越重要，越需要优先调度。所以任务的优先级由周期决定，与周期成反比。任务的周期越短优先级越高；任务的周期越长，优先级越低。假设同时有两个任务 τ_1 和 τ_2 被触发，如果 $T_1<T_2$，τ_1 的周期小于 τ_2，则 $P_1>P_2$，τ_1 的优先级大于 τ_2，τ_1 将先于 τ_2 得到系统调度。在任务的截止时间等于周期的假设条件下，Liu 和 Layland 已经证明 RM 算法是最优的静态优先级调度算法并给出 RM 算法可调度性判定的充分条件：

$$U=\sum_{i=1}^{n}\frac{C_i}{T_i}\le n(2^{\frac{1}{n}}-1) \tag{1}$$

RM 算法还常常与其它算法（如 FIFO 和 RR 等）结合使用，构造成效果更好的实时调度方式。

3.1.4 LLF 算法

LLF（Least Laxity First 最低松弛度优先）调度算法属于动态优先级调度算法，也是一种截止时间驱动的调度算法[31]。LLF 算法关注任务的松弛度。任务的松弛度是指假设任务立即在不可抢占条件下执行完毕到截止时间之间的剩余时间，可以用 L

来表示，L=D-E-t（t 表示当前系统时间）。松弛度表明了任务可以往后延迟执行的时间。松弛度越小，任务可以延迟的程度越小。当松弛度为 0 时，任务需要立即调度否则就不能在截止时间之前完成；如果松弛度为负值，任务将出现超时。LLF 算法根据任务松弛度的大小分配优先级，松弛度越小，优先级越高，每次调度时都选取松弛度最小的任务来执行。LLF 的可调度条件与 EDF 一样，CPU 使用率也可达 100%，也是一种最优的动态优先级调度算法。但是 LLF 在调度时会引起大量的任务切换，并且在有两个任务的松弛度一样时会产生松弛环，造成系统死锁。系统将在这两个任务之间不停的切换直到松弛环解除。另外由于系统需要不断地计算松弛度，开销较大。这些原因使得 LLF 算法并不实用，应用较少。

3.1.5 EDF 算法

EDF 调度算法（Earlist Deadline First 也称为截止时间驱动算法 DDS）是一种截止时间驱动的调度算法[32]。EDF 算法根据任务截止时间的大小决定任务调度的优先级。任务的截止时间越小，表明任务需要执行的紧迫程度越高，分配到的优先级也越高。这里所讨论的截止时间是指任务的绝对截止时间 d。d 等于任务的到来时间加上相对截止时间，即 d=A+D。因此任务的优先级并不是事先确定的，而是在系统运行过程中由任务的到来时间动态决定。同属于一个周期的子任务中，先到达的任务将比后到达的任务具有更高的优先级。每次系统调度时，EDF 算法都从任务集中选取截止时间最小的任务来执行。

在截止时间等于周期的条件下，Liu 和 Layland 证明了 EDF 算法可调度性判断的充分必要条件。

$$U = \sum_{i=1}^{n} \frac{C_i}{T_i} \le 1 \tag{2}$$

从公式(2)可以看出 EDF 算法对 CPU 利用率比较高，可达到 100%。实际上这个冲要条件是在理想模型下讨论的，截止时间等周期，所有任务同时到达，不考虑非周期任务影响等其它因素。于是有很多学者对更为复杂的模型进行了研究，并给出了判定方法。J.W.S 针对一般 EDF 系统提出了可调度条件判定的充分条件

$\Delta = \sum_{i=1}^{n} \frac{C_i}{\min\{D_i, T_i\}} \le 1$。Leung 和 Merrill 提出在时间段[0,max{s_i}+2H]内所有任务的截止时间都得到满足，那么任务集是可调度的，其中 min{s_i}=0，H 是所有任务周期的最小公倍数。Baruah 将任务扩展到非周期任务，并给出了可调度条件判定的充分必要条件。这些方法更贴近实际模型，但是复杂度太高，实现较难。所以本文将在理想模型下讨论 EDF 算，并使用公式(2)做为可调度条件判定的依据。

RM 算法实现简单、系统开销小，在现实中得到了广泛的应用，但是其 CPU 利用率不高[33]。LLF 算法系统开销较大，在嵌入式实时操作系统中并不适用。EDF 算法是一种最优的动态优先级的调度算法，效果介于 RM 算法和 LLF 算法之间。下面将对 EDF 算法做详细介绍。

3.2 EDF 算法分析

由于要动态的计算优先级，实现 EDF 算法的系统开销较大。另外 EDF 算法在系统过载时调度性能急剧下降。这些都在一定程度上制约了 EDF 算法的应用。下文将从算法实现的系统开销、处理器利用率和处理系统过载的能力等方面对 EDF 进行详细分析和讨论。

3.2.1 实现 EDF 的系统开销

实现 EDF 算法的系统开销包括调度开销和运行开销。

调度开销是指与调度相关操作的开销，与系统的调度算法和进程的实现方式有关[34]。一般来说，静态优先级调度算法的调度开销要小于动态优先级调度算法。如果使用的静态优先级的调度算法，比如 RM 算法，由于优先级是固定的，可以根据具体情况把优先级数确定在一个合适的值，然后为每个优先级维护一个任务队列。每个任务队列再和 FIFO 、RR 算法结合。这样任务队列维护的操作时间是确定的，复杂度为 O(1)，系统的调度开销较小。如果使用 EDF 算法，由于任务的截止时间都不一样，由任务的到来时间动态决定。系统必须维护一个按任务截止时间由大到小排列的任务队列。当有新任务到达或者任务离开时，系统需要将此任务加入或者删除。这样，当任务数 n 值很大时，花在维护任务队列的时间就变多，调度开销就变

得很大。与 RM 算法相比，EDF 算法的调度开销较大。

运行开销是指任务在运行时的开销，这里主要是指任务切换引起的系统开销。任务切换的次数越多，系统开销越大；任务切换的次数越少，系统开销越少。任务切换与系统调度算法的抢占有关系。使用 EDF 算法引起的任务抢占要比静态优先级算法 RM 少[35]。这是因为 RM 算法中的任务优先级固定，高优先级任务的到来总会抢占低优先级的任务。即便后到来的高优先级的任务在等待正在执行的低优先级的任务执行完后才执行也不会超过时限，RM 算法仍然会让高优先级的任务抢占低优先级的任务。而在 EDF 算法中考虑的是任务的绝对截止时间。任务的绝对截止时间等于任务到来的时间加上任务的相对截止时间。这样后到来的任务往往比先到来的任务有更低的优先级，不会抢占先到来的任务。因而 EDF 算法中任务的切换次数要比 RM 算法少，运行开销要小。下面以实例说明。

表 3.1 任务集 S_1

任务	周期	执行时间
任务 1	30	10
任务 2	60	30

假设有一个任务集 S_1={ τ 1, τ 2, }，S1 中有两个任务，每个任务的周期和执行时间如表 3.1 所示。

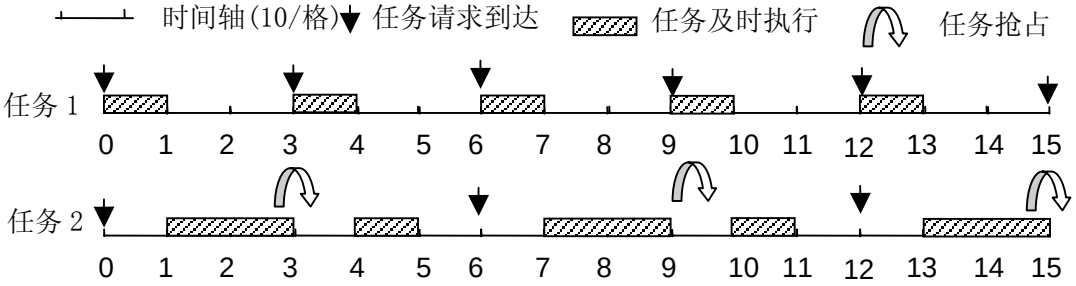

图 3-1 RM 算法调度任务集 S_1

使用 RM 算法调度效果图 3-1 所示。由于任务 1 的周期小于任务 2，任务 1 的优先级较高。在调度过程中，任务 2 的所有请求在执行时由于任务 1 的到来被抢占，引起系统任务切换。

使用 EDF 算法调度效果如图 3-2 所示。任务 2 的请求在执行时，有任务 1 到来，比较任截止时间发现任务 1 的截止时间与任务 2 正在执行的请求一样，但是由于任

务 1 后到达，优先级低，并未抢占任务 2 的执行，没有引起系统的任务切换。

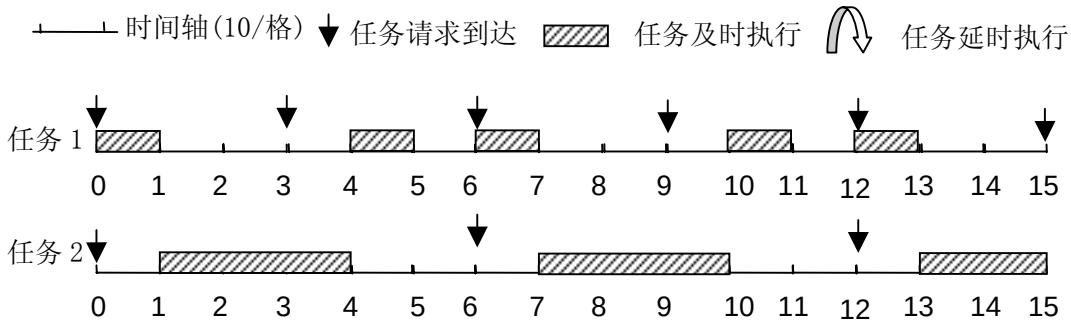

图 3-2 EDF 算法调度任务集 S_1

3.2.2 EDF 的 CPU 利用率

EDF 算法的 CPU 利用率上届值为 1，这可以从其可调度条件判定的充分必要条件公式(2)中看出。也就是说 EDF 算法能够充分地利用处理器资源，没有浪费。而且 EDF 算法的 CPU 利用率与任务数没有关系。无论任务数多少，CPU 利用率上界值都为 1。

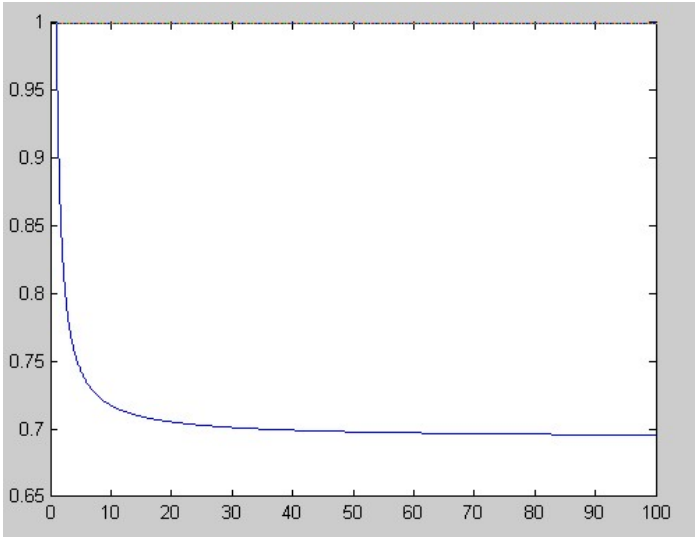

图 3-3 RM 算法可调度任务负载与任务个数关系

RM 算法的 CPU 可利用率根据公式(3)等于 $n(2^{\frac{1}{n}}-1)$，与任务数有关系。如图 3-3 所示。当 n 为 1 时，利用率上界为 100%；当 n 为 2 时，利用率为 $2(2^{\frac{1}{2}}-1)\approx 0.83$。随着任务数 n 的增多，上届值逐渐趋向于 ln2=0.69。RM 算法的 CPU 利用率较小（但是在任务周期、截止时间和执行时间存在一些特殊关系时，可大于此函数确定的上

界值[36])。

因此 EDF 算法的 CPU 利用率比 RM 算法高，可以承受更多的系统负载[37]。一些不能被 RM 算法调度的任务集可以被 EDF 算法调度。这是 EDF 算法相对于最优的静态优先级调度算法 RM 的最大优点。下面举例说明。

表 3.2 任务集 S_2

任务	周期	执行时间
任务 1	30	10
任务 2	40	20
任务 3	60	10

假设有一个任务集 S1={ τ 1, τ 2, τ 3}，S1 中有三个任务，每个任务的周期和执行时间如表 3.2 所示（时间长度用所占的系统时间片个数表示）。任务集的 CPU 使用率 U=10/30+20/40+10/60=1。分别按照 RM 算法调度和 EDF 算法调度该任务集，假设所有任务的第一个请求是同时到达。

用 RM 算法调度效果如图 3-4 所示，周期较短的任务 1 和任务 2 都在时限内完成，而周期较长的任务 3 有部分请求未能在规定的时限内完成，出现了延迟。这是因为任务集的CPU资源需求U大于RM算法此时所能调度的上届值 $0.78(3\times(2^{\frac{1}{3}}-1)\approx 0.78)$。

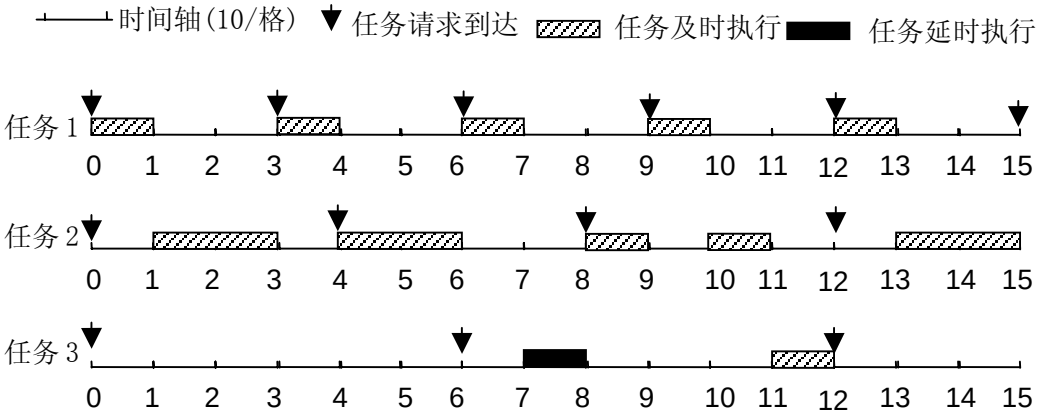

图 3-4 RM 算法调度任务集 S_2

如果使用 EDF 算法调度。效果如图 3-5 所示。所有任务的实时性都得到了满足。这是因为任务集 U=1，在 EDF 算法可调度范围内。

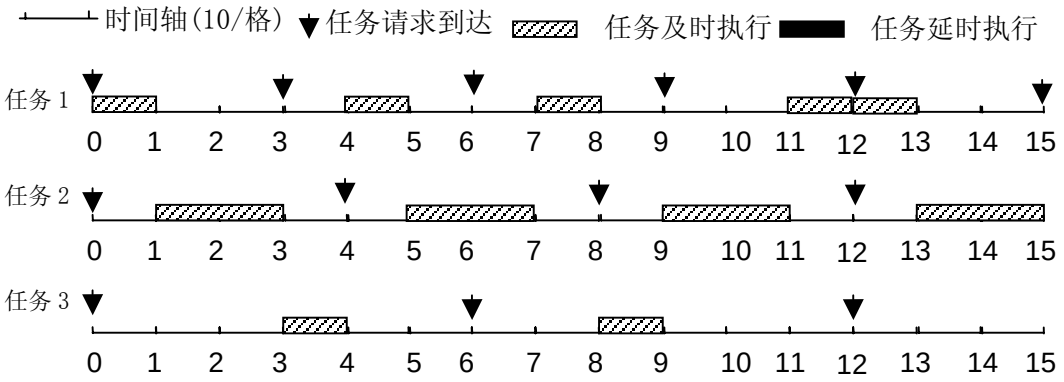

图 3-5 EDF 算法调度任务集 S_2

3.2.3 EDF 处理系统过载的能力

EDF 算法对处理器有较高的利用率，但是处理系统过载的效果却不佳。下面用实例分析 EDF 算法处理系统过载的能力，并与 RM 算法作比较。假设有一个任务集 $S_1=\{\tau_1, \tau_2, \tau_3\}$，$S_1$ 中有三个任务，每个任务的周期和执行时间如表 3.3 所示。分别用 RM 算法和 EDF 算法调度。

表 3.3 任务集 S_3

任务	周期	执行时间
任务 1	30	20
任务 2	40	20
任务 3	60	10

RM 算法调度效果如图 3-6 所示。短周期任务 1 的实时性得到满足，次短周期任务 2 在任务 1 的空闲时间里才得到执行，出现了延迟。而任务 3 由于处理器的时间已经被任务 1 和任务 2 使用完了没有空余而得不到调度。

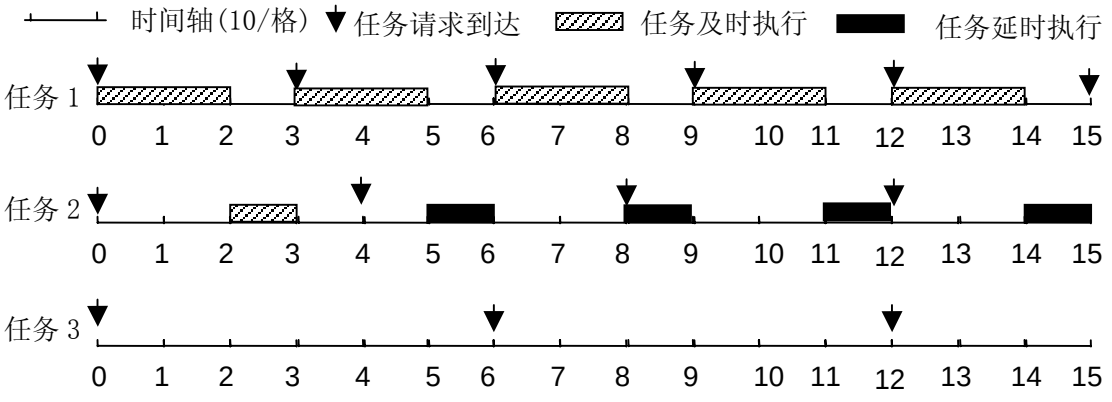

图 3-6 RM 算法调度任务集 S_3

这是因为 RM 算法是以任务周期长短来确定优先级的静态优先级调度算法，任务的周期一般是固定的，优先级也是确定的。系统每次都调度优先级最高的任务运行。所以最高优先级的任务总会先得到处理器资源，实时性可以保证。剩余低优先级的任务只能在处理器空闲时调度，实时性无法保证。

用 EDF 算法调度的效果如图 3-7，所有任务的实时性都得不到满足，只有开始的几个请求按时完成，之后的请求都出现了延迟。在系统过载情况下，EDF 算法的调度效果不如 RM 算法。

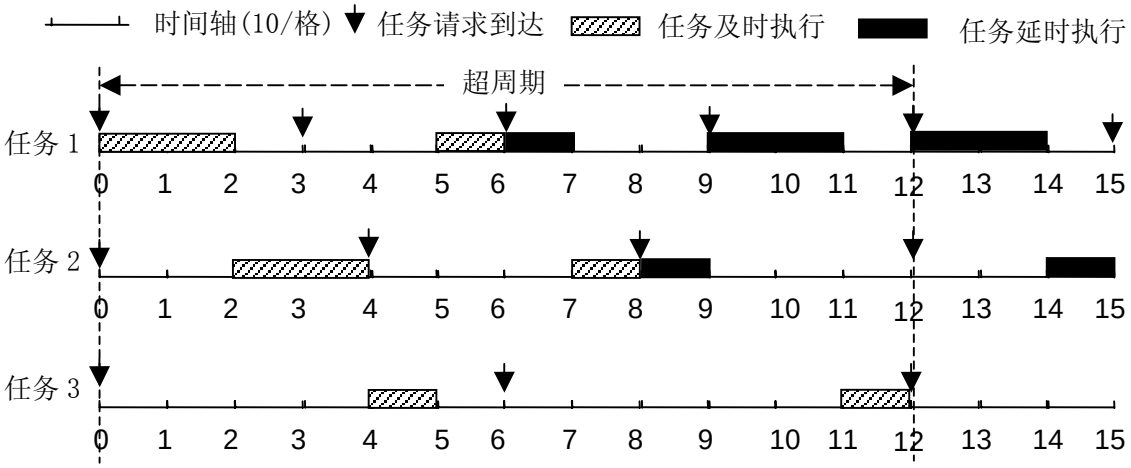

图 3-7 EDF 算法调度任务集 S_3

EDF 算法分析复杂些，为方便理解，引入超周期[38]的概念。对于有 n 个任务的任务集，超周期是所有任务周期的最小公倍数，即 $T=[\tau_1, \tau_2, \tau_3, \tau_4 \cdots \tau_n]$。。假设每个超周期开始时，所有任务的请求同时到达。由于超周期是每个任务周期的公倍数，超周期结结束时，所有任务的周期刚好也结束。任务在每个超周期中重复着与其它超周期相同的到达时间和顺序。也就是说整个任务集可以看成是一个以超周期为更大周期的任务。因此只要分析 EDF 算在一个超周期中的表现就可以了。

系统在一个超周期中提供的可用处理器时间是固定的。但是在系统过载情况下任务集对处理器的需求时间大于可利用的处理器时间。因此多出来的需求时间必须在下一个超周期处理。由于 EDF 算法是以任务请求的绝对截止时间为优先级，先到达的任务请求因为到达时间比较早，其相对截止时间也比较早，优先级要比后到达的任务请求高。也就是说先到达的任务请求总体来说要在后达到的任务请求之前得

到调度。这样在一个超周期中，系统必须先处理上一个超周期中未得到响应的处理器需求。一个超周期未能调度的任务请求就必须占用后一个超周期的可用处理器时间。如本例第一个超周期中的 $\tau_{1,4}$ 和 $\tau_{2,3}$ 未能在本超周期内分到处理时间，于是占用了第二个超周期的处理器时间。引起后一个超周期用于处理本超周期任务的时间相应的减少，进而造成本超周期内更多的任务请求未能及时响应。最终形成一个“多米诺骨牌”效应[39]，使得后续大量任务请求的实时性都得不到满足。

3.3 EDF 优点和缺点

通过上述分析可以知道 EDF 算法在调度的灵活性、运行开销和 CPU 利用率上比静态调度算法有优势，但是在调度开销和处理系统过载时的性能方面不如静态调度算法，具体结论如下。

EDF 算法优点：

（1）灵活性和适应性好，EDF 算法与静态优先级算法比较最大的优点在于任务优先级由系统动态决定的。这种优先级主要取决于两个因素：一是任务要求的相对截止时间，反应了任务自身对时限的要求；二是任务请求的到达时间，反应了任务对时限的客观要求。这种动态优先级决定机制比静态优先级更能反应任务的实际需求。

（2）运行开销小。EDF 以任务的截止时间来决定优先级。在保证时限的情况下，尽量减少任务的抢占和任务切换较少，系统运行开销比较小。

（3）CPU 使用率高，EDF 算法在保证任务集实时性的同时充分利用 CPU，可以使 CPU 利用率达 100%。而且 CPU 的利用率上界值与具体的任务数无关。这一点，EDF 算法比 RM 算法好。RM 算法 CPU 利用率比较低，而且随着任务数的增多而降低。

EDF 算法缺点

（1）系统调度开销大。因为 EDF 算法任务的优先级是由系统根据任务的截止时间动态决定，而任务的截止时间与任务的周期、到来时间、执行时间和相对截止时间有关系。所以系统必须使用一些资源去存储和计算这些相关信息。另外 EDF 算

法的任务队列管理也比较复杂，与具体的任务数有关系，并随着任务数的增多而增大。因此相对于静态优先级调度算法，EDF 算法的调度开销要大一些。

（2）优先级的决定因素还不够全面。EDF 算法只考虑任务的截止时间，并没有考虑任务本身的价值。在实际应用中，有些价值比较大的任务应该比价值小的任务得到优先调度，尤其在系统过载时，但是 EDF 算法对这些任务一视同仁，并没有区分。

（3）EDF 算法不能处理过载。当任务集的实时处理要求大于处理器的处理能力时，EDF 算不能进行适当的取舍保证部分任务的实时性，而是仍然按照原有的优先级调度，造成一种”多米诺骨牌效应”,引起大部分实时任务的超时。这一点 EDF 算不如静态优先级调度算法。

3.4 EDF 算法改进

EDF 算法在系统轻载情况下表现出较好的性能，但是不能分辨任务的重要性、系统的调度开销大，影响了其在实际中的广泛应用。另外 EDF 算法不能处理系统过载，使得 EDF 算只适用于软实时环境，不适合硬实时系统环境。下面将对 EDF 算进行一些改进，使其尽量满足系统的硬实时要求。

3.4.1 EDF 算法改进思路

对与 EDF 算法的改进主要从以下几个方面考虑：如何让系统分辨重要的任务与不重要的任务；如何减小系统的调度开销；如何在系统过载的情况下有选择的保证部分任务的实时要求等。具体描述如下：

（1）给任务增加一个优先级，用于表示该任务的重要程度[40]。在实际应用中，任务的实时要求分为软实时和硬实时要求。其中有硬实时要求的任务往往是重要的任务，如数据采集和处理，这些任务必须在规定时限内完成，否则会出现信息丢失影响系统结果。有软实时要求的任务重要程度不及有硬实时要求的任务，往往可以稍后一些调度，比如音频和视频任务等。EDF 算法决定优先级的机制过于单一，仅仅依靠每个任务请求的绝对截止时间来，并不考虑任务是否重要，不能很好的反映实际要求。可能会出现在任务截止时间相同的情况下，一个软实时要求的任务得到

调度，而另一个硬实时要求的任务得不到调度。因此可以引入一个表示任务重要性的优先级，用这个优先级和任务的绝对截止时间来共同决定任务调度时的优先级。这样，当两个任务的截止时间相同时，有硬实时要求的任务就会比有软实时要求的任务先得到调度。

（2）当系统过载时，在系统的调度能力范围内选择几个重要任务执行，保证重要任务的实时要求，把“多米诺骨牌效应”给所有任务带来的影响局限在部分不重要的任务中[41,42]。EDF 算法最大的缺点就是在系统过载时不能满足大多数任务的实时要求。既然不能满足所有任务的实时要求，不如有选择的满足部分任务的实时要求。一种处理办法就是选择最重要的几个任务，它们的 CPU 使用率之和小于等于 1，然后用 EDF 算法对这几个任务进行调度。仅在 CPU 有空闲时才对剩余的任务集进行 EDF 调度。对于同时有硬实时要求和软实时要求的系统可以这样考虑，系统过载时如果只有硬实时任务或者同时有硬实时和软实时任务则可以先对最重要的几个硬实时任务进行 EDF 调度，在 CPU 空闲时再对次要的硬实时任务和软实时任务进行 EDF 算法调度；如果系统中只有软实时任务则挑选几个最重要的软实时任务进行 EDF 调度。这种处理方式既利用了 EDF 算法 CPU 使用率高的特点，又保证了系统过载时重要的任务可以得到保证，适用于有硬实时要求的环境。

（3）算法实现时通过选择合适的数据结构和组织方式来减小调度开销。EDF 算法比静态优先级算法的开销大主要在于：需要存储与时间相关的信息和动态地计算任务优先级。EDF 算法需要知道任务的周期、截止时间、执行时间和到达时间等，由于这些信息存在一定的关系，在一些特殊情况下可以互相计算出来，而且一些系统所提供的时钟机制可以为这些信息的存储提供便利，所以在算法实现时可以根据具体情况尽量选择存储较少信息的方式。另外在系统选择调度线程方式和任务队列的组织管理方面，也可根据具体情况设计。系统可以在任务到来时就根据计算的截止时间来确定任务调度调度顺序，然后把任务放置到队列的相应位置，这样在调度时就不需要额外的计算了；也可以先简单地把任务放置到队列中，在需要调度时再进行计算。考虑到系统调度频率还是比较高，如果调度时计算，系统开销会比较大，所以建议选择在任务到来时就计算。

3.4.2 改进后 EDF 算法描述

改进后的 EDF 算法具体描述如下:

每个任务到来时需要提供周期、截止时间、执行时间和表示该任务重要程度和紧迫程度的优先级等信息，其中任务优先级值越小，优先级越高表明该任务越重要。

为每个优先级准备一个队列，用于存放本优先级的就绪任务，任务在队列中按照绝对截止时间由小到大排列。对于截止时间相同的任务，优先级高的任务应位于优先级低的任务前面。建立一个数组，数组的每个元素指向其下标所对应的任务优先级队列，可称之为优先级队列数组[43]。就绪任务到来时先根据优先级找到对应的优先级队列，在根据截止时间加入该优先级队列中。每个优先级队列的队首任务就是本优先级中绝对截止时间最小的任务。

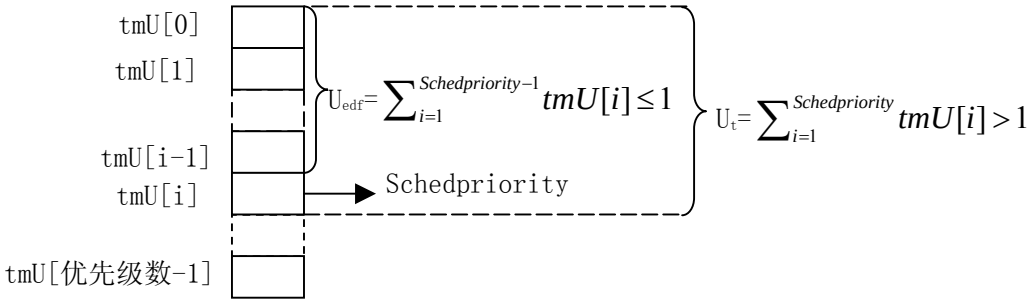

图 3-8 tmU[优先级数]、U_{edf}、schedpriority 和 U_t 关系

在系统中定义变量 tmU[优先级数]、U_{edf}、Schedpriority 和 U_t 用于标示系统任务集中 EDF 算法可以调度的子任务集。tmU[优先级数]用于记录每个优先级的所有任务 CPU 利用率之和。在系统轻载情况下，Ut 是系统中所有优先级的就绪任务的 CPU 利用率之和，即 $U_t=\sum_{i=1}^{优先级数} tmU[i]$，n 为系统中所有任务的个数。系统过载情况下，Ut、Uedf 和 Schedpriority 满足的关系如图 3-8 所示。其中 $Uedf=\sum_{i=1}^{Schedpriority-1} tmU[i]\le 1$ 并且 $\sum_{i=1}^{Schedpriority} tmU[i]>1$。Schedpriority 表示 EDF 算法可调度子任务集的优先级界限。优先级小于 Schedpriority 的任务可参与系统调度，优先级大于 Schedpriority 的任务不可参与系统调度。优先级等于 Schedpriority 的任务根据该任务的 CPU 使用率 u 与 U-Uedf 的关系决定，当 u≤U-Uedf 时可以参与系统调度；当 u>U-Uedf 时不参

与系统调度。当有新的任务到达或者需要从任务队列中删除任务时，系统需要重新计算 Uedf 和 Schedpriority。具体方法是：按优先级从高到低累加每个优先级的所有任务的 CPU 使用率，当累加值大于 1 时停止累加。Schedpriority 等于此时的优先级。Uedf 等于累加值减去 tmU[Schedpriority]。计算 Uedf 和 Schedpriority 实际上是在对任务集进行可调度性判定。在综合考虑系统开销和判定效果后，本算法使用公式(2)作为可调度性的判定依据。此外随着时间的流逝，一个任务的某次请求在可调度性判定之前没有得到调度，由于其剩余时间减小，此请求的紧迫程度增大，此刻的时限要求类似于周期为其剩余时间的周期任务（算法假设任务的截止时间等于周期）。所以在进行可调度性判定时使用此任务请求的剩余时间代替周期，即 $u' = \frac{C_i}{d-t}$ (t 为系统当前时间)。

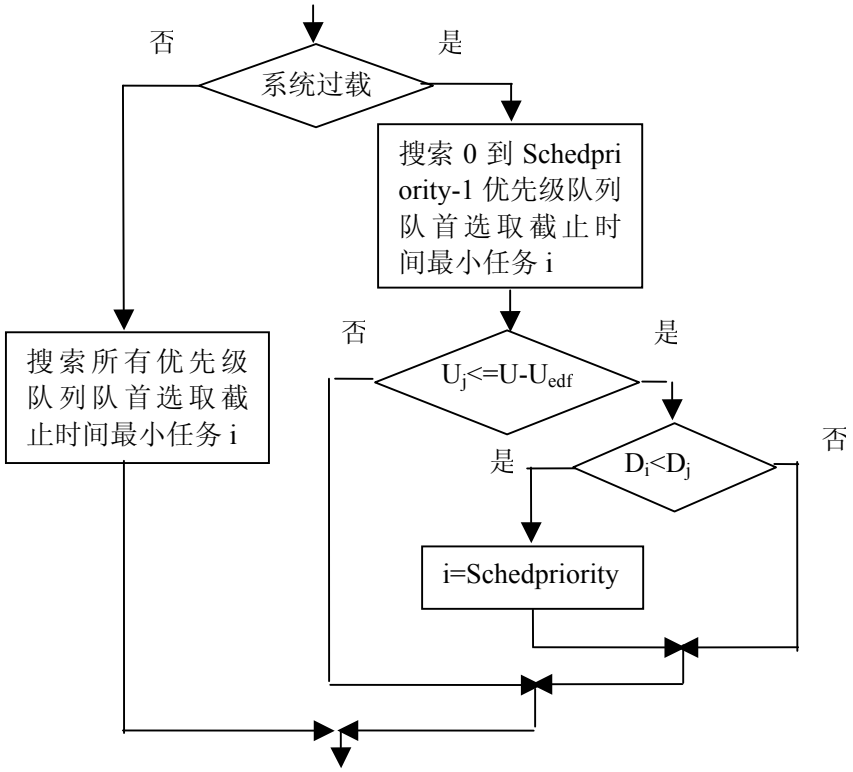

图 3-9　改进后 EDF 算法

系统调度时选择下一个目标线程的算法如图 3-9 所示。

输入：系统所有线程；

输出：下一个将要调度的线程。

1．首先判断系统的状态，如果 U 小于等于 1，转步骤 2；如果如果 U 大于 1，转步骤 3。

2．系统处于轻载状态下，系统任务集可以用 EDF 算法合理调度。于是搜索所有优先级队列的队首任务，选取截止时间最小的任务 i 调度，算法结束。

3．系统处于过载状态下，系统的任务集不能使用 EDF 算法调度，所以应该把 Uedf 和 Schedpriority 确定的可调度子任务集做为 EDF 算法调度的对象。在这个子任务集中选择截止时间最小的线程调度。即搜索小于 Schedpriority 的所有优先级任务队列的队首任务，选取截止时间最小的任务 i。

4．判断 Schedpriority 优先级队列的队首任务 j 是否在可调度的子任务集内，如果不在子任务集内，转步骤 5；如果在则转步骤 6。

5．选择前面步骤 2 选出的任务 i 作为下一个调度线程。

6．比较任务 j 和选出的任务 i 的截止时间大小，选取值小的线程作为下一个将要调度的线程。

下面对算法的正确性进行证明：

1．系统轻载情况下，所有任务的处理器利用率$U=\sum_{i=1}^{n}\frac{C_i}{T_i}\le 1$，根据 EDF 算法的可调度性判定条件公式（2）可知任务集可以用 EDF 算法调度。

2．系统过载情况下，对于所选出的可调度任务子集有$U'=\sum_{i=1}^{m}u_i'=\sum_{i=1}^{m}\frac{C_i}{d-t}\le 1$，由于$d-t\le T$，所以有$\frac{1}{d-t}\ge\frac{1}{T}$。即$\sum_{i=1}^{m}\frac{C_i}{T_i}\le U'\le 1$。同理根据公式（2）可知该任务集可以用 EDF 算法调度。

表 3.4 增加优先级后的任务集 S_3

任务	周期	执行时间	优先级
任务 1	30	20	3
任务 2	40	20	2
任务 3	60	10	1

使用改进后的 EDF 算调度 3.2.3 中系统过载时的任务集。首先给每个任务增加

一个表示重要程度的优先级，如表 3.4 所示(优先级值越小越高)。

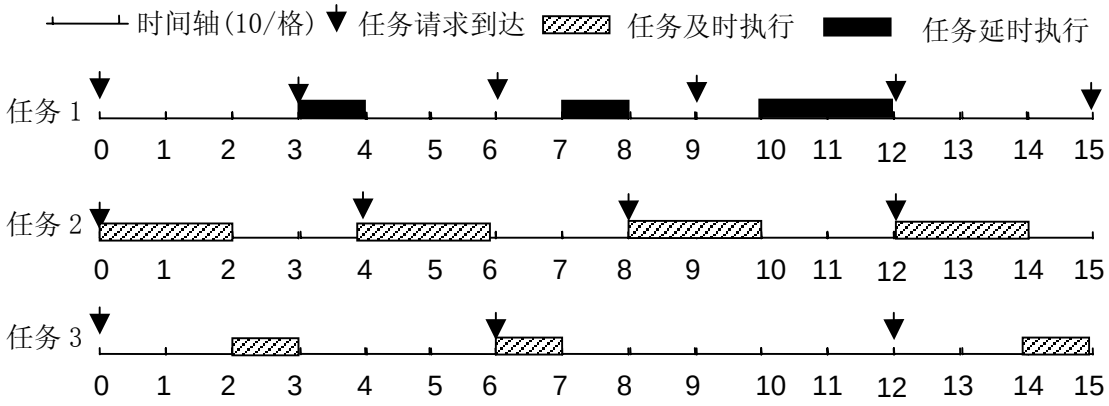

图 3-10　改进后 EDF 算法调度任务集 S_3

使用改进后 EDF 算法调度过程如图 3-10 所示。在系统过载情况下优先级高的任务 2 和 3 的实时性得到了满足，优先级低的任务 1 仅仅在处理器调度完高优先级任务有空余时得到执行。

3.4.3 改进后 EDF 算法复杂度

EDF 算法只需要维护一个与任务数相关的队列，所以空间复杂度为 $O(n)$。改进后 EDF 算法把任务按照优先级分类，需要为每个优先级维护一个队列，每个队列需要一个队列指针，所以空间复杂度等于 $O(n)+O(m)$（其中 m 为优先级数目）。

EDF 算法在调度时，直接选取任务队列的队首线程作为下一个调度对象，所以 EDF 算法调度的时间复杂度等于 $O(1)$。在维护任务队列上，EDF 算法需要根据任务的截止时间搜索队列，时间复杂度为 $O(n)$。

考虑改进后 EDF 算法的时间复杂度。算法将就绪任务以优先级队列数组的形式组织，每个优先级队列的第一个任务总是此优先级队列中截止时间最短的，调度时只需要搜索每个优先级队列的队首线程，选取截止时间最短的线程。系统轻载时，搜索长度为优先级数；系统过载时，搜索长度缩小为 0 到 Schedpriority。因此调度最大搜索长度为系统的优先级个数与具体任务数无关，是一个定值，时间复杂度为 $O(1)$，符合实时系统的要求。在系统任务队列的管理上，算法需要将就绪任务加入相应的优先级就绪队列中，搜索的长度只与本优先级的任务数有关。故管理队列的时间复杂度为 $O(n)$，　n 为就绪队列中任务的个数。

3.5 小结

本章从实现开销、CPU 利用率和处理系统过载的能力具体分析了 EDF 算法，并找出其优点和不足；针对 EDF 算法的缺陷提出了一些改进方法。EDF 算法是一动态的优先级调度算法，灵活性和适应性好、运行开销小、CPU 利用率高。但是由于需要存储和计算一些与时间相关的信息以及较为复杂的任务队列维护，EDF 算法的调度开销比较大。可以通过在算法实现时结合系统具体情况采用合理的数据结构和组织方式减小开销。另外 EDF 算法的优先级决定机制过于单一，仅用任务的截止时间来确定优先级，不能体现任务的重要性。因此可以考虑引入一个表示任务重要程度的因子来和截止时间共同决定任务的优先级。在系统过载的情况下，EDF 算法调度效果不佳，产生的“多米诺效应”会造成大多数任务超时。可以通过选择几个比较重要的任务来调度的方法改善 EDF 在系统过载下的调度效果。

4 改进 EDF 在 ARTs-OS 中的实现

本章以嵌入式实时操作系统 ARTOS 为应用平台，将改进后的 EDF 算法和其已有的调度算法相结合作为系统的新调度方式，实现了对实时和非实时任务的有效调度。下面将介绍如何在 ARTs-OS 实现 EDF 算法。首先添加几个辅助变量、修改一些内核数据结构，主要是与线程和调度相关的数据。然后是线程队列的组织和调度器函数接口的编写。

4.1 相关变量和数据结构

在线程控制块的调度信息中增加 period、dealine、execute、realtype 和 u 这几个变量。其中 period 表示任务的周期、dealine 表示任务的相对截止时间、execute 表示任务的执行时间。这些信息在线程创建时用户提供或者系统指定,并利用 ARTs-OS 系统中的时钟粒度 10 毫秒作为时间单位，以方便与系统中的时间滴答数做计算。u 表示该任务的 CPU 使用率，在线程创建时由系统根据任务的周期和执行时间计算。由于 ARTs-OS 已实现多种调度方式，realtype 用于表示使用何种调度方式，线程在创建时必须通过参数告知系统是否参与 EDF 算法调度。

另外在系统内核的调度模块中增加几个全局变量 tmU[NBPRIORITY]、U、U_{edf}、schedPriority。tmU[NBPRIORITY]用于记录每个优先级的所有任务的 CPU 利用率之和，NBPRIORITY 为系统中任务的优先级个数 96。U 用于判断系统是否过载，以便采取不同的处理方式。在系统轻载情况下，U 是系统中所有就绪任务 CPU 利用率之和，值应该小于等于 1；在系统过载情况下，$U=U_{edf}$+tmU[schedPriority]，值应该大于 1。U_{edf}表示优先级 0~schedPriority-1 的所有任务 CPU 利用率之和。schedPriority 是 EDF 算法可调度的优先级的界限。优先级大于该值的任务不参与调度；优先级小于该值的任务参与调度；优先级等于该值的任务，从前往后累计该优先级任务队列中每个任务的 CPU 利用率之和，在满足累计之和加上 U_{edf} 小于等于 1 的任务集可以参与调度，剩余的不能参与调度。每当有新线程就绪或者有线程从就绪转为其它状态时，系统都需要重新计算这些变量的值。计算的伪代码如下：

```
BEGIN
MAXVALE→time
0→Uedf
For(0→i;i<96;i++){
    0→tmU[i]
        If i 优先级队列不为空{
            i 优先级队列的队首任务→elem
            while(elem 不为空){
    if(elem 任务还没超过截止时间) tmU[i]+i 任务的执行时间/i 任务的剩余时间
→tm[i]
    else tmU[i]+1→tmU[i]
    elem-＞next→elem
}
Uedf+ tmU[i] →Uedf
        }
        Uedf→U
        If(Uedf>1){
Uedf- tmU[i] →Uedf
i→schedPriority
return i
}
}
i→schedPriority
return I
END
```

4.2 线程队列组织方式

EDF 算法的线程队列组织方式如图 4-5 所示。对于就绪线程，建立一个优先级队列数组 edfList[95]；对于阻塞线程，建立一个阻塞队列 edfblockList。当线程已经

获得足够资源在等待处理器运行时将其放入就绪队列中；当线程因为要等待资源、消息或者事件到来时，将其放如阻塞队列。edfList[95]是一个指针数组，每个元素指向与其下标相对应优先级队列。每个优先级都有一个优先级队列，线程在对应的优先级队列里按照线程的绝对截止时间由小到大排列。为方便向前和向后搜索，优先级队列采用双向链表结构。队列还设置有两个指针，分别指向队列首部和尾部。就绪线程到来时，系统应根据线程的优先级从 edfList[95]中找到对应的优先级队列，再根据线程的截止时间加入到该优先级队列中。阻塞队列只是暂时存放线程，所以可简单的从队列首部或者尾部将线程加入，为方便删除和查找，阻塞队列也使用双向链表结构。另外 ARTs-OS 系统的线程控制块中有一个 hint 指针，可以用于指向该线程在线程队列中的位置。这样当需要从队列中删除该线程时，可以直接根据 hint 值操作，而不需要搜索整个队列。

线程创建时，系统应该根据线程是否参与 EDF 算法调度将其放入不同的队列中。对于使用 EDF 算法调度的线程，将线程加入 EDF 的线程队列中；对于不参加 EDF 算法的线程，系统将其加入 ARTs-OS 系统原有的线程队列。

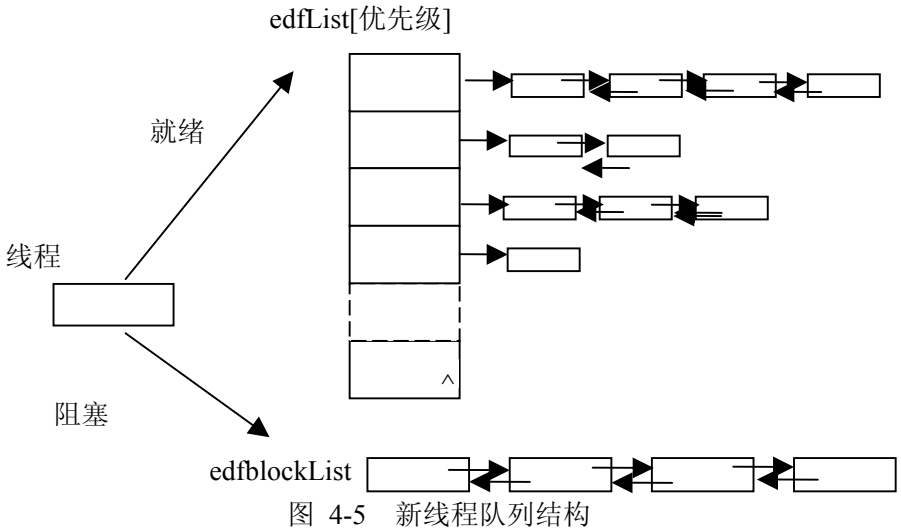

图 4-5　新线程队列结构

4.3 EDF 调度器对象实现

创建一个调度器对象 sched_edf，实现里面的接口函数，如图 4-6 所示。由于线程队列是系统的公共资源，各个线程都可以访问和操作，为保持内核的同步和数据

一致性，创建一个 SchedLock 锁。线程只有在获得 SchedLock 锁后才可以对相应的数据操作。

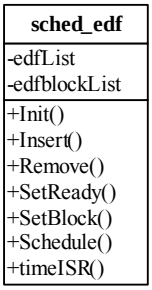

图 4-6　EDF 调度器

Init()是调度器的初始化，在系统任务模块初始化时调用。主要完成一些公共变量的初始化和公共资源的准备。包括优先级队列数组 edfList [95]和阻塞队列 edfblockList 的初始化；U、U_{edf}、tmU[NBPRIORITY]、schedPriority 等的赋初值；SchedLock 锁资源的准备。

Insert()在线程创建时调用，用于将线程控制块放入阻塞队列。线程在创建时需要做一些资源准备工作和进行参数设置，并将线程控制块放入阻塞队列。首先获取 SchedLock 锁资源，然后设置线程状态为阻塞，将新线程的控制块从尾部加入阻塞队列 edfblockList 中。最后释放 SchedLock 锁资源。

Remove()在线程销毁时调用，用于将线程控制块从就绪队列或者阻塞队列中除去。首先判断线程的状态，如果是阻塞状态，从阻塞队列 edfblockList 中删除线程控制块；如果是就绪状态，从优先级队列 edffirst[线程优先级]中删除线程控制块。释放 SchedLock 锁资源。

SetReady()用于将线程控制块放入就绪队列中。首先检查线程的状态是否是运行或者就绪，如果是，则表明线程已经在就绪队列中，函数返回。否则根据线程的优先级在 edfList [96]中找到该优先级所对应线程队列的指针。搜索此队列，根据线程的绝对截止时间大小将线程加入到此优先级队列中。然后重新计算系统中 U、Uedf 和 schedPriority 的值。其中将线程加入就绪队列的伪代码如下：

BEGIN

新线程→addelem

新线程优先级→priority

edflist[priority]→elem

if(edflist[priority]队列为空){

设置 addelem 为 edflist[priority]队列的第一个元素，并将 edflist[priority]队列的队首和队尾都指向 addelem

}

While(elem 不为空){

If(elem 的截止时间大于 addelem)break;

If(elem 的截止时间等于 addelem 并且 elem 的优先级低于 addelem)break;

}

If(elem 为 edflist[priority]的队首元素)

将 addelem 加入到 elem 前面,设置其为 edflist[priority]的队首元素。

else if(elem 为空)

将 addelem 加入到 edflist[priority]的队尾，设置其为 edflist[priority]的队尾元素

else 将 addelem 加入到 elem 前面。

END

SetBlock()函数用于将线程放入阻塞队列。首先检查线程的状态，如果已经是阻塞状态表明线程已经在阻塞队列中，函数返回。否则将线程控制块从线程所在的优先级队列 edffirst[线程优先级]中删除，然后将其加入到阻塞队列 edfblockList 的尾部。最后重新计算系统中 U、Uedf 和 schedPriority 的值。

Schedule()函数是系统调度的核心函数，在每次调度时执行，用于选择下一个运行的线程。首先判断系统状态，如果系统处于轻载情况下，则搜索所有优先级队列的队首线程，选取截止时间最小的线程运行；如果系统处于过载情况下，只有 0 到 Schedpriority 优先级队列中的线程参与调度，搜索 0 到 Schedpriority-1 优先级队列的队首线程，选取截止时间最小的线程 i。然后判断 Schedpriority 优先级的线程队列是否为空。若为空调度线程 i；否则判断该队列的队首线程 j 是否参与系统调度。比较其 CPU 使用率 u_j 与 U-Uedf 的大小。当 u_j>U-Uedf 时,线程 j 不参与系统调度,系统调度 i 线程。当 u_j<=U-Uedf 时，线程 j 参与系统调度。将线程 i 的截止时间 D_i 与线程

j 的截止时间 D_j 进行比较。如果 $D_i < D_j$ 调度线程 i；如果 $D_i > D_j$ 调度线程 j。 其伪代码如下：

```
BEGIN
MAXVALE→time
NULL→item
NULL→item1
If(U>1){
    For(0→i;i<Schedpriority;i++){
        If(i 优先级队列 edflist[i]不为空){
            If(edflist[i]队首任务截止时间大于 time){
                edflist[i]队首任务→item
                edflist[i]队首任务截止时间→time
            }
        }
    }
    If(edflist[Schedpriority]不为空){
        edflist[Schedpriority] 队首任务→item1
        if(Uedf+item1 的 CPU 利用率>1)return item
        else{
            if(item 的截止时间小于 items 的截止时间)return item1
            else return item
        }
    }
}
Else {
    For(0→i;i<96;i++){
        If(i 优先级队列 edflist[i]不为空){
            If(edflist[i]队首任务截止时间大于 time){
```

```
                edflist[i]队首任务→item
                edflist[i]队首任务截止时间→time
            }
        }
        Return item
}
Return NULL
END
```

timeISR()是与时间片相关的调度函数，在每个时间片结束时调用。一些需要根据时间片进行相关操作的算法比如 RR 算法需要实现它。EDF 算法中不需要实现它。

sched_edf 调度器实现完成后将 ARTs-OS 中的调度参数设置为 sched_edf 即可以在 ARTs-OS 启动 EDF 调度方式。

4.4 小结

本章描述了如何在 ARTs-OS 系统中实现 EDF 调度算法，并通过具体实验对算法的改进方式进行了验证。为实现 EDF 算法，在 ARTs-OS 系统中构建一个优先级队列数组用于按优先级存放就绪线程，相同优先级的线程按照截止时间从小到大排列。再设置变量 U 和 U_{edf}，当有就绪线程需要加入和删除时，根据可调度性判定方法计算法 U 和 U_{edf}，用于标识 EDF 算法的可调度任务集。最后按照 EDF 算法改进后的思路编写 EDF 线程调度器。

5 改进 EDF 算法的性能测试与结果分析

下面将对在 ARTs-OS 系统中实现的改进后的 EDF 算法进行测试。通过具体实验结果分析改进后 EDF 算法处理过载的能力并给出结论。

5.1 测试环境与方法

首先引入截止期满足率的概念。截止期满足率用于表示调度算法中任务被满足的程度[44]。当任务在截止期限内完成时称该任务的截止期得到满足，如果任务在截止时间超过后仍然没有完成则称该任务的截止期未得到满足。周期任务的截止期限满足率等于截止期限得到满足的子任务数比上子任务总数，截止期满足率越高表明该任务时限得到满足的程度越高，截止期满足率越低表明该任务时限得到满足的程度越低。在 ARTs-OS 系统中分别使用 EDF 算法和改进后的 EDF 算法在系统轻载和过载情况下调度任务集，比较任务集的截止期限满足率。

实验硬件平台：ARM9 处理器、200MHz、32M 内存；

软件平台：ARTs-OS 系统。

测试方法：启动三个周期任务，执行 1200 个系统时间片，计算各个任务的截止期限满足率。

5.2 测试结果与分析

（1）系统轻载情况下两种算法的任务截止期限满足率：

表 5.1　系统轻载下任务集 S_4

任务	周期	执行时间	优先级
任务 1	30	10	1
任务 2	40	10	2
任务 3	60	10	3

假设有一个任务集 $S_4=\{\tau_1, \tau_2, \tau_3\}$，S1 中有三个任务，每个任务的周期、执行时间和优先级如表 5.1 所示. 此时任务集 CPU 利用率=10/30+10/40+10/60≈0.75<1,

系统处于轻载状态下。分别用 EDF 算法和改进后的 EDF 算法调度，测量每种调度算法下任务截止期得到满足的个数。

使用 EDF 算法测量结果如表 5.2 所示。三个任务的截止期满足任务数都等于各自的总任务数，截止期满足率都为 100%。EDF 算法可以对此任务集进行有效调度。

表 5.2 任务集 S_4 按照 EDF 算法调度的结果

任务	总任务数	截止期满足任务数	截止期满足率
任务 1	40	40	100%
任务 2	30	30	100%
任务 3	20	20	100%

使用改进后 EDF 算法测量结果如表 5.3 所示。三个任务的截止期满足任务数也都等于各自的总任务数，截止期满足率都为 100%。改进后 EDF 算法可对此任务集进行有效调度。

表 5.3 任务集按照改进后 EDF 算法调度结果

任务	总任务数	截止期满足任务数	截止期满足率
任务 1	40	40	100%
任务 2	30	30	100%
任务 3	20	20	100%

因此在系统轻载情况下，EDF 算法和改进后的 EDF 算法都保证任务的有效调度。

（2）系统过载情况下两种算法的任务截止期限满足率：

表 5.4 系统过载下任务集 S_5

任务	周期	执行时间	优先级
任务 1	30	20	1
任务 2	40	10	2
任务 3	60	10	3

假设有一个任务集 $S_5=\{\tau_1, \tau_2, \tau_3\}$，$S_5$ 中有三个任务，每个任务的周期、执行时间和优先级如表 5.4 所示.此时任务集 CPU 利用率=20/30+10/40+20/60=1.25>1,系统处于过载状态下。分别用 EDF 算法和改进后的 EDF 算法调度，测量每种调度算法下任务截止期得到满足的个数。

使用 EDF 算法测量结果如表 5.5 所示。每个任务都只有 1 个任务的截止期限得

到满足，截止期满足率比较低。

表 5.5　任务集 S_5 使用 EDF 算法调度的结果

任务	总任务数	截止期满足任务数	截止期满足率
任务 1	40	1	2.5%
任务 2	30	1	3.3%
任务 3	20	1	5%

使用改进后 EDF 算法测量结果如表 5. 6 所示。任务 1 和任务 2 的截止期得到满足的任务数比较接近总任务数，截止期满足率较高，而任务 3 的截止期均得不到满足，截止期满足率等于 0%。这是因为在系统处理器资源不能满足需求时，改进后的 EDF 算选取了优先级比较高的任务 1 和任务 2 组成可调度子集，首先保证这两个任务的时限，在处理器空闲时才调度任务 3。另外可以发现理论上任务 1 和任务 2 可以用 EDF 算法调度，但实际上这两个任务的时限要求并不是完全得到满足的。因为系统运行需要占用一定的处理器时间，有系统开销。所以实际用于任务的处理器时间要比理论上少一些。本实验中，系统开销除了有实现 EDF 算法的开销，还有模拟周期任务的开销。如果在 EDF 算法进行可调度性判断时考虑系统时间（比如中断处理时间[45]、线程队列管理时间等），调度效果会更好。

表 5.6　任务集 S_5 使用改进后 EDF 算法调度的结果

任务	总任务数	截止期满足任务数	截止期满足率
任务 1	40	35	87.5%
任务 2	30	22	73.3%
任务 3	20	0	0%

通过上述实验结果可以看出在系统轻载情况下，EDF 算法和改进后的 EDF 算法都能使任务的截止期限满足率得到满足。而在系统过载的情况下，EDF 算法下大多数任务的截止期限都得不到满足，截止期限满足率较低；改进后的 EDF 算法保证了在可调度范围内重要的任务的截止期限，达到了算法改进的目的。

5.3 小结

本章对改进后的 EDF 算法进行了测试和分析。在 ARTs-OS 系统中分别使用 EDF 算法和改进后的 EDF 算法对不同负载下的任务集进行调度。实验结果表明：在轻负

载情况下，EDF 算法和改进后的 EDF 算法都能对任务集进行有效调度；在重负载情况下，EDF 算法不能保证任务集的有效调度，改进后的 EDF 算法能够在可调度范围内保证重要任务的有效调度。

6 结束语

随着人们在生产、生活中对实时处理需求的不断增多，嵌入式实时操作系统的应用越来越广泛。如何根据实际需要，在具体的平台为用户提供最好的实时服务一直是业内技术人员关注的焦点。实时调度算作为提高嵌入式实时操作系统的重要因素也备受关注。经过多年的研究和探索，实时调度算法无论是在理论还是实际应用上都取得了不少成果。比如将静态优先级调度算法 RM 算法与 FIFO、RR 相结合的混合调度算法可以得到较理想的调度效果，已经被许多嵌入式实时操作系统所采用。但是还有许多理论也很优秀的算法由于各种局限未能应用到实际中去。本文就是针对这种现象对实时调度算法进行了研究和讨论，具体工作可总结如下：

（1）详细地分析了 ARTs-OS 的实时调度机制。ARTs-OS 采用了微内核结构、可动态加载和配置内核，ARTs-OS 采用了多进程多线程模型，使用了基于优先级的可抢占调度方式，并提供高精度的时钟支持。

（2）从系统开销、CPU 利用率和处理系统过载的能力方面具体分析了 EDF 算法。因为需要进行一定量的计算和维护任务队列，EDF 的调度开销比较大。但是由于调度时充分利用了任务的截止时间，避免了一些不必要的任务抢占，EDF 调度算法的系统开销比较小。EDF 算法的处理器利用率较高，理想状态下可达 100%。在系统过载情况下，EDF 算法会产生“多米诺效应”，造成大量任务的时限得不到满足。

（3）针对 EDF 算法的不足，提出一些改进方法并给出具体算法描述。引入一个表示任务重要程度的因子与任务的截止时间共同决定任务的优先级。算法具体实现时，应该根据系统的具体特点采用合理的数据结构减小系统开销。系统过载情况下， 按照任务的重要程度选取 EDF 算法可以调度的任务子集做为调度对象，优先保证重要任务的实时性。

（4）在 ARTs-OS 系统中实现了改进后的 EDF 算法，并通过具体测试检验 EDF 算法的改进方法。实验结果表明改进后的 EDF 算法在系统轻载状态下，调度效果与 EDF 算法一致；在系统过载状态下，能够保证重要的任务优先得到调度。

由于时间和笔者水平所限，本文对 EDF 算法的讨论还有如下不足尚待完善和研

究：

（1）EDF 算法的可调度性判定是实现 EDF 算法的关键。本文在实现 EDF 算法时使用的是理想模型下的简单判定方法，尽管有根据实际情况稍作修改，但是判定效果仍然不理想。EDF 算法可调度性判定的效果与所考虑的影响因素和所使用的实时调度模型有关。所考虑的影响因素越多，实时调度模型越复杂，可调度性判定效果越好，但是需要的算法复杂度越高。现有理论中，在考虑了非周期任务、任务截止时间和周期不存在特定关系等因素后，EDF 算法的可调度性判定算法常常是指数级的计算量，并不适合实际应用，尤其是资源比较宝贵的嵌入式实时操作系统。因此如何通过简单地计算准确地判定 EDF 算法下的任务集是否能够调度有待进一步的研究和讨论。

（2）对 EDF 算法的改进上存在算法性能退化。在系统过载情况下，如果所有任务都是相同优先级的将排列在同一个队列中。那么此算法就和 EDF 算法一样会产生“多米诺效应”，不能处理系统过载情况。具体实现方面，算法在调度时需要搜索所有可调度优先级队列的队首元素。当有优先级队列为空时，就产生不必要的开销，这种开销会随着系统优先级数目的增大而增多。尽管这种开销可以确定在一个范围内，但还是在一定程度上影响了系统的实时性能，应当尽量减小。因此如何一步提高 EDF 算法处理系统过载的能力和减小系统开销仍需进一步分析和探索。

致 谢

在论文完成之际我要向所有关心和帮助过我的人表示感谢。

首先要感谢我的导师刘云生教授。刘老师学识渊博、理论功底丰厚、工作认真严谨，常常结合自己的经历和经验来教导我们，督促我们不断改进自己，让我受益匪浅。感谢刘老师在百忙之中抽空指导本文，对本文提出了许多宝贵的意见。

感谢实验室项目组的两位组长。是他们让我有机会接触到了许多新的领域，学习到了许多新知识，同时他们不断专研和精益求精的精神一直影响着我。

感谢实验室项目组的同学。与大家一起讨论研究、互相学习、互相帮助的过程让我感受到了集体温暖和集体协作的重要性。

感谢我的同班好友。在研究生的日子里，她们的真挚友谊和热情帮助让我的研究生生活变得丰富多彩。

感谢我的亲人在我遇到困难的时候不断的支持和鼓励我，让我有勇气去克服困难。

参考文献

[1] 张荫沛，徐国治，周玲玲．微内核操作系统在嵌入式平台上的应用．电子产品世界，2009，3：44-46

[2] Fengxiang Zhang, Alan Burns. Schedulability Analysis for Real-Time Systems with EDF Scheduling. IEEE Transactions on computers, 2009, 58: 1250-1258

[3] 翟小艳，徐家品．基于 UC/OS-II 任务调度算法的改进与分析．微计算机信息，2010，26(23)：63-65

[4] 王豫，谷建华. 两种主流嵌入式实时操作系统的研究. 微处理机，2009，1：124-127

[5] 熊江．三种嵌入式操作系统的分析与比较．单片机与嵌入式系统应用，2003，5：14-17

[6] 季志均，马文丽，陈虎等．四种嵌入式实时操作系统关键技术分析．计算机应用研究，2005，9：4-8

[7] 李庆诚，顾健．嵌入式实时操作系统性能测试方法研究．单片机与嵌入式系统应用，2005，8：19-21

[8] 雷红卫，桑楠，熊光泽．嵌入式实时操作系统中断管理技术研究．单片机与嵌入式系统应用，2004，5：16-19

[9] 曾非一，桑楠，熊光泽．嵌入式系统内存管理方案研究．单片机与嵌入式系统应用，2005，1：5-7

[10] 王霞，马忠梅，何小庆等. 提高嵌入式 Linux 时钟精度的方法. 计算机工程，2006，32(23)：70-72,96

[11] 田小华，陆少华．嵌入式操作系统的时钟机制的研究．计算机与数字工程，2009，37(1)：69-70,114

[12] Ali H. Dogru, Murat M. Tanik. A Process Model for Component-Oriented Software Engineering. IEEE Software, 2003: 34-41

[13] Armen Zakarian, Andrew Kusiak. Analysis of Process Models. IEEE Transactions on Electronics Packaging Manufacturing, 2000:137-147

[14] 李京，段汕．Linux2.6 内核的实时调度的研究．电脑知识与技术，2007，23：

1361-1363

[15]张杰，阳富民，卢炎生等．EDF 统一调度硬实时周期任务和偶发任务的可调度性判定算法．小型微型计算机系统，2009，12：2383-2388

[16]安阳，常明，张琳等．嵌入式 Linux 系统实时进程调度算法改进．计算机与信息技术，2009，Z1：49-52

[17]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)：12-14

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

[19]童立靖．实时系统的自适应进程调度方法研究：[博士学位论文]．北京：中国科学院研究生院（软件研究所），2004．

[20]王涛，刘大昕．多处理器单调速率任务分配算法性能评价．计算机科学，2007，1：272-277

[21]Baruah S K, Gehrke J E, Plaxton C G. Fast Scheduling of Periodic Tasks on multiple resources. Parallel Processing Symposium, 1995: 280-288

[22]Gardner M K, Liu J W S. Performance of Algorithms for Scheduling Real-time System with Overrun and Overload. Real-Time Systems, 1999:287-286

[23]Zhao Wei, Ramamritham Krithi, Stankovic John A. IEEE Transactions on Computers, 1987: 949-960

[24]Ching-Chin Han, Kwei-Jay Lin, Chao-Ju Hou. Distance-Constrained Scheduling and Its Application to Real-Time Systems. IEEE Transctions on Computers, 1996: 814-826

[25]N. Audsley, A. Burns, M.Richardson, K. Tindell et al. Applying new scheduling theory to static priority pre-emptive scheduling, 1993: 284-292

[26]Marco Spuri, Giorgio C. Buttazzo. Efficient Aperiodic Service under Earliest Deadline Scheduling. IEEE, 1994:2-11

[27] Ching-Chih Han, Hung-ying Tyan. A Better Polynomial-Time Schedulability Test for Real-Time Fixed-Priority Scheduling Algorithms. IEEE, 1997: 36-45

[28] Sylvain Lauzac, Rami Melhem, Daniel Mosse. An Improved Rate-Monotonic Admission Control and Its Applications. IEEE Transactions on Computers, 2003, 3: 337-350

[29] Almut Burchard, Yingfeng Oh, Sang H. son. New Strategies for Assigning Real-Time Tasks to Multiprocessor Systems. IEEE Transanctions on Computers, 1995: 1429-1442

[30] 谢健平．单处理器环境下实时混合任务的调度算法研究．[硕士学位论文]．武汉理工大学，2008

[31] Sung-Heun Oh, Seung-Min Yang. A Modified Least-Laxity-First scheduling algorithm for real-time tasks. Real-Time Computing Systems and Applications, 1998:31-36

[32] Theodore P. Baker. An Analysis of EDF Schedulability on a Multiprocessor. IEEE Transactions on Parallel and Distributed System, 2005, 16: 760-768

[33] John Lehoczky, Liu Sha, Ye Ding. The Rate Monotonic Scheduling Algorithm: Exact Characterization And Average Case Behavior. IEEE, 1989: 166-171

[34] 吕鸣松．嵌入式工业监控系统中实时调度策略的研究：[硕士论文]．沈阳：东北大学，2005．

[35] 王永吉，陈秋萍．单调速率及其扩展算法的可调度性判定．软件学报，2004，6：799-814

[36] Buttazzo GC. Rate Monotonic vs. EDF: Judgment day. Real-Time Systems, 2005, 29(1): 5-26

[37] 邢群科，郝红卫，温天江．两种经典实时调度算法的研究与实现．计算机工程与设计，2006，27(1)：117-123

[38] 涂刚．软实时系统任务调度算法研究：[博士论文]．武汉：华中科技大学图书馆，2004．

[39] 余祖峰，蔡启先，刘明．EDF 调度算法的实时改进．广西工学院报，2010，21(1)：82-85

[40] 王永炎，王强，王宏安．基于优先级表的实时调度算法及其实现．软件学报，2004，

15(3)：360-369

[41]李岩，钟兆君．在 Linux 下改进 EDF 实时调度算法．中国新技术新产品，2009，5：23

[42]王昊，张钟澍．一种改进的 Linux 实时进程调度算法－RAD 算法．成都信息工程学院学报，2009，24(3)：219-223

[43]洪伟，苏晓龙，王香婷．Linux 系统实时调度策略的研究与实现．微计算机信息，2010，26(6-1)：165,207-209

[44]洪雪玉，张凌，袁华．Linux 下的实时调度算法．华南理工大学学报（自然科学版），2008，36(4)：104-109

[45]Kevin Jeffay, Donald L. Stone. Accounting for Interrupt Handling Costs in Dynamic Priority Task Systems. IEEE, 1993:212-221