广东工业大学

硕士学位论文

基于μC/OS-II的实时多任务调度算法的研究

姓名：张波

申请学位级别：硕士

专业：控制理论与控制工程

指导教师：邓则名;高军礼

20090501

摘要

随着实时嵌入式系统应用的日益复杂化，系统可能要处理多个实时任务，同时各个任务之间也可能有多种信息传递。如果仍采用原来的程序设计方法将存在两个问题：一是中断可能得不到及时响应，处理时间过长，这对于一些控制场合是不允许的；二是系统任务多，要考虑的各种可能也多，各种资源如调度不当就会发生死锁，降低软件可靠性。

在实时系统中，系统的正确性不仅仅依赖于计算的逻辑结果而且依赖于结果产生的时间。从这个角度上看，可以把实时系统定义成“一个能够在指定或者确定的时间内完成系统功能和对外部或内部、同步或异步时间做出响应的系统”。

实时系统主要面向与时间因素相关的应用需求，相应的处理过程必须在规定的时间限制内完成。此外，实时系统中多任务共享资源很容易出现无限优先级反转现象，最终导致系统崩溃。因此，任务调度与资源管理是实时系统最重要的组成部分。

本文选择实时内核作为研究对象，以嵌入式微处理器S3C44B0为硬件平台。对μC/OS-II操作系统的移植进行研究，改造μC/OS-II实时内核，将其移植到微处理器S3C44B0中。进而研究和解决实时系统中实时调度算法和任务管理的问题。μC/OS-II实时内核以抢占的方式调度任务，并且源代码开放。它为每个任务分配唯一的优先级，不支持相同优先级任务的调度，而在实际应用中为相同功能的任务分配不同的优先级不是一个很好的逻辑设计。因此本文扩展了μC/OS-II实时内核，使其支持相同优先级的轮转调度，从而实现了μC/OS-II中固定优先级抢占和同优先级单调速率调度方式相结合的一种混合调度策略。通过实验验证，扩展后的μC/OS-II实时内核能够支持相同优先级任务的单调速率调度，并且仍然能够保持μC/OS-II抢占式内核的特点。最后应用RapidRMA工具对调度结果进行分析。

关键词： 实时系统，任务调度，优先级反转，优先级继承

ABSTRACT

With the complex application of real-time embedded system, a system maybe deal with multitask and each task may transmit information to each other. If using old design method, there are two questions: the one is interrupt will not be responded in time and deal time is long, this is forbidded in some control; the other is that if resources are scheduled unsuitably dead lock will happen and it may drop the reliability of system, because the system has multitask and many possibilities to think.

In real-time system, the correct of the system relies not only on the calculation of the logic result, but also on the time when the result occurs. From this point, we can give the definition of real-time system: a system that can complete system function and respond to inner or external, synchronization or asynchronous time in specify time.

Real-time system mainly faces to application needs related to time factor. Responsible deal process must complete in prescriptive time. Otherwise, in the system, sharing of resources by multi tasks can bring priority inversion and lead to crash of the system. So task schedule and resource manage are the most important constitutions to real-time system.

The article chooses real-time kernel as research object and embedded microprocessor S3C44B0 as hardware platform. Reconstruct μC/OS-II real-time kernel and plant it in microprocessor S3C44B0. Research and solve scheduling algorithms and task manage problems in real-time system. μC/OS-II real-time kernel schedules tasks using preemptive method and the source code is open. It assigns uniquely priority to every task and don't support scheduling of the same priority. In real application, it is not a good logic design assigning different priority to same function task. So the article expands the μC/OS-II real-time kernel, makes it support schedule of same priority. Realize a mix scheduling method in μC/OS-II combining fixed priority preemptive and RM. By testing, expansion μC/OS-II real-time kernel can support RM algorithms to same priority tasks and keep preemptive characteristic of μC/OS-II. Final use RapidRMA analyzing the schedule result.

KeyWords: Real-time System, task scheduling, priority inversion, priority ceiling

独创性声明

秉承学校严谨的学风与优良的的科学道德，本人声明所呈交的论文是我个人在导师的指导下进行的研究工作及所取得的研究成果。尽我所知，除了文中特别加以标注和致谢的地方外，论文中不包含其他人已经发表或撰写过的研究成果，不包含本人或其它用途使用过的成果。与我一同工作的同志对本研究所做的任何贡献均已在论文中作了明确的说明，并表示了谢意。

本学位论文成果是本人在广东工业大学读书期间在导师的指导下取得的，论文成果归广东工业大学所有。

申请学位论文与资料若有不实之处，本人承担一切相关责任，特此声明。

论文作者签字：

指导教师签字：

2009 年 6 月 1 日

第一章 绪论

1.1 研究背景及意义

随着电子技术的迅速发展，特别是随着大规模集成电路产生而出现的微型计算机给人类生活带来了根本性的改变。如果说微型机的出现使现代科学研究得到了质的飞跃，那么可以毫不夸张的说，嵌入式微控制器技术的出现则是给现代工业控制领域带来了一次新的技术革命。在微型计算机中微处理器是现代信息技术的“心脏”，微处理器主要分为通用式和嵌入式两大类。

嵌入式系统则被定义为：以应用为中心，以计算机技术为基础，软件硬件可裁剪，适应应用系统，对功能、可靠性、成本、体积、功耗严格要求的专用计算机系统[1]，嵌入式系统是将先进的计算机技术、半导体技术、电子技术和各个行业的具体应用相结合后的产物。嵌入式系统是面向用户、面向产品、面向应用的。

嵌入式处理器的功耗、体积、成本、可靠性、速度、处理能力、电磁兼容性等方面均受到应用要求的制约。和通用计算机不同，嵌入式系统的硬件和软件都必须高效率地设计，量体裁衣、去除冗余。力争在同样的硅片面积上实现更高的性能，对芯片配置进行裁剪和添加才能达到理想的性能，这样才能在具体应用对处理器的选择面前更具有竞争力。

嵌入式系统被广泛应用于生物医学仪器、智能汽车、通信设备、网络设备、仪器仪表、手持设备等诸多领域。嵌入式操作系统则是嵌入式系统应用中的核心。嵌入式系统是计算机硬件和软件的结合体，或许还加上机械等其他部分，被设计来完成专门的功能。在一些情况下，嵌入式系统是一个大的系统或产品的一部分，就像汽车上的防抱死装置，与通用计算机相对。最初的嵌入式系统是不带操作系统的，只是用来完成某一个特定的单一功能。随着软硬件技术的发展，完成单一功能的嵌入式系统已经不能适应市场的需要，因此出现了带操作系统的嵌入式系统。

目前嵌入式应用领域的一个发展趋势是采用实时操作系统。随着应用的复杂化，一个嵌入式控制器系统可能要处理很多实时任务，同时各个任务之间也可能

有多种信息传递。如果仍采用原来的程序设计方法将存在两个问题：一是中断可能得不到及时响应，处理时间过长，这对于一些控制场合是不允许的；二是系统任务多，要考虑的各种可能也多，各种资源如调度不当就会发生死锁，降低软件可靠性。

从性能上讲，实时内核和普通的操作系统的区别主要在于对实时性的要求不一样。在实时计算中，系统的正确性不仅仅依赖于计算的逻辑结果而且依赖于结果产生的时间。从这个角度上看，可以把实时系统定义成“一个能够在指定或者确定的时间内完成系统功能和对外部或内部、同步或异步时间做出响应的系统”[2]。

如果应用项目对额外的需求可以承受，应该考虑使用实时内核。这些额外的需求是：内核的价格、额外的ROM和RAM开销，以及2%到4%的CPU额外负荷。但是实时内核通常价格较高，在一些低成本的应用中，就无法使用实时内核。

1.2 实时操作系统发展概况

虽然嵌入式系统是近几年才开始真正风靡起来的，但事实上嵌入式这个概念却很早就已经存在了，从上个世纪 70 年代单片机的出现到今天各种嵌入式微处理器、微控制器的广泛应用，嵌入式系统少说也有了近 30 年的历史。纵观嵌入式系统的发展历程，大致经历了以下四个阶段[3]：

(1)无操作系统阶段

嵌入式系统最初的应用是基于单片机的，大多以可编程控制器的形式出现，具有监测、伺服、设备指示等功能，通常应用于各类工业控制和飞机、导弹等武器装备中，一般没有操作系统的支持，只能通过汇编语言对系统进行直接控制，运行结束后再清除内存。这些装置虽然已经初步具备了嵌入式的应用特点，但仅仅只是使用 8 位的 CPU 芯片来执行一些单线程的程序，因此严格地说还谈不上“系统”的概念。

这一阶段嵌入式系统的主要特点是：系统结构和功能相对单一，处理效率较低，存储容量较小，几乎没有用户接口。由于这种嵌入式系统使用简便、价格低廉，因而曾经在工业控制领域中得到了非常广泛的应用，但却无法满足现今对执行效率、存储容量都有较高要求的信息家电等场合的需要。

(2)简单操作系统阶段

20 世纪 80 年代，随着微电子工艺水平的提高，IC 制造商开始把嵌入式应用中所需要的微处理器、I/O 接口、串行接口以及 RAM、ROM 等部件统统集成到一片 VLSI 中，制造出面向 I/O 设计的微控制器，并一举成为嵌入式系统领域中异军突起的新秀。与此同时，嵌入式系统的程序员也开始基于一些简单的“操作系统”开发嵌入式应用软件，大大缩短了开发周期、提高了开发效率。

这一阶段嵌入式系统的主要特点是：出现了大量高可靠、低功耗的嵌入式 CPU（如 Power PC 等），各种简单的嵌入式操作系统开始出现并得到迅速发展。此时的嵌入式操作系统虽然还比较简单，但已经初步具有了一定的兼容性和扩展性，内核精巧且效率高，主要用来控制系统负载以及监控应用程序的运行。

(3)实时操作系统阶段

20 世纪 90 年代，在分布控制、柔性制造、数字化通信和信息家电等巨大需求的牵引下，嵌入式系统进一步飞速发展，而面向实时信号处理算法的 DSP 产品则向着高速度、高精度、低功耗的方向发展。随着硬件实时性要求的提高，嵌入式系统的软件规模也不断扩大，逐渐形成了实时多任务操作系统（RTOS），并开始成为嵌入式系统的主流。

这一阶段嵌入式系统的主要特点是：操作系统的实时性得到了很大改善，已经能够运行在各种不同类型的微处理器上，具有高度的模块化和扩展性。此时的嵌入式操作系统已经具备了文件和目录管理、设备管理、多任务、网络、图形用户界面（GUI）等功能，并提供了大量的应用程序接口（API），从而使得应用软件的开发变得更加简单。

(4)面向 Internet 阶段

21 世纪将是一个网络的时代，将嵌入式系统应用到各种网络环境中去也将是一种必然。目前大多数嵌入式系统还孤立于 Internet 之外，随着 Internet 的进一步发展，以及 Internet 技术与信息家电、工业控制技术等的结合日益紧密，嵌入式设备与 Internet 的结合才是嵌入式技术的真正未来。

信息时代和数字时代的到来，为嵌入式系统的发展带来了巨大的机遇，同时也对嵌入式系统厂商提出了新的挑战。嵌入式系统的研究和应用产生了如下新的显著变化：

1)新的微处理器层出不穷，嵌入式操作系统自身结构的设计更加便于移植，能够在短时间内支持更多的微处理器。

2)嵌入式系统的开发成了一项系统工程，开发厂商不仅要提供嵌入式软硬件系统本身，同时还要提供强大的硬件开发工具和软件支持包。

3)通用计算机上使用的新技术、新观念开始逐步移植到嵌入式系统中，如嵌入式数据库、移动代理、实时 CORBA 等，嵌入式软件平台得到进一步完善。

4)各类嵌入式操作系统迅速发展，常用的嵌入式操作系统有 Windows CE、Palm、μClinux、pSOS、Nucleus、VxWorks 和 μC/OS-II 等。

5)网络化、信息化的要求随着 Internet 技术的成熟和带宽的提高而日益突出，以往功能单一的设备如电话、手机、冰箱、微波炉等功能不再单一，结构变得更加复杂，网络互联成为必然趋势。

6)精简系统内核，优化关键算法，降低功耗和软硬件成本。

7)提供更加友好的多媒体人机交互界面。

1.3 研究内容

本论文的选题来源于广东省重大战略产业发展专项资金项目《基于光纤通信总线的多轴联动高档数控系统研发及产业化》的子项目的“基于 SERCOS-III 的数控装备网络通信系统的研究与应用”。

本文的研究重点以一个源码公开的实时嵌入式操作系统 μC/OS-II 为基础，主要研究和解决实时系统中多任务实时调度算法的问题。根据其内核优先权分配的特点，对其任务调度算法的函数进行一定的改进。μC/OS-II 中无法调度相同优先级任务，使得它不适应某些实时应用，对优先级反转问题的解决方法也有缺陷。因此本文在这些方面做了相关的改进工作，这部分也是论文的重点。对 μC/OS-II 实时操作系统的可移植性进行了研究，将改进后的内核移植到嵌入式微处理器 S3C44B0，用它来实现系统多任务的调度控制和管理。

主要包括以下四个方面：

(1)主要介绍实时嵌入式系统的内核分类以及任务调度算法。对实时调度理论和方法进行分析和比较，着重讨论针对单一任务类型的调度方法和针对各种类型实时任务的混合调度策略。

(2)对 μC/OS-II 实时操作系统中共享资源管理的问题进行研究，分析 μC/OS-II 实时内核中优先级反转问题产生的原因和对系统实时性的影响。可以利用优先级置顶和优先级继承两种实现方法抑制 μC/OS-II 中的优先级反转，并对两种实现

方法的性能进行分析、比较。

(3)μC/OS-Ⅱ操作系统移植研究。μC/OS-Ⅱ的大部分是用标准 C 编写的，只含有一小部分汇编代码，有很强的可移植性。μC/OS-Ⅱ代码包括三部分：核心代码部分、应用相关部分和处理器相关部分。在移植时需要改造的就是应用相关部分和处理器相关部分。应用相关部分包括 os_cfg.h 和 includes.h。可根据应用的需要对它们进行设置以对 μC/OS-Ⅱ进行裁减和配置。处理器相关部分包括 os_cpu.h、os_cpu_a.asm 和 os_cpu_c.c。

(4)多任务调度研究。包括：任务的确定、任务优先级分配、多任务调度方法研究等。对实时嵌入式操作系统任务调度算法函数进行改进，提出优先权调度算法和单调速率调度算法相互配合的混合调度算法。用 RAPIDRMA 对任务调度结果进行仿真。

本文共分五章，各章内容安排如下：

第一章 绪论：主要介绍了课题的研究背景和意义，以及实时系统发展的概况。对主要的研究内容作了说明。

第二章 实时系统相关理论研究：本章首先介绍了实时系统的概念、特点及应用领域。然后介绍了 Linux、VxWorks、QNX、WindowsCE、μC/OS-II 等实时操作系统，通过对比选定了本文所采用的操作系统——μC/OS-II 操作系统。介绍了实时操作系统的概念、体系结构、衡量标准和实时内核，分析了实时内核的特点，并以图形的形式剖析了抢占式内核和非抢占式内核的工作原理。最后研究了实时系统中的调度问题。包括调度的概念、实时任务的分类及常见的任务模型；从针对单一任务调度算法和各种类型任务的混合调度算法，对调度现状进行了分析研究。

第三章 μCOS-Ⅱ实时操作系统研究：本章从 μC/OS-Ⅱ相对于其他操作系统的优点和特点入手，主要对其内核结构进行了分析研究。重点介绍了任务管理模块，对用户任务结构、任务状态、任务控制块和任务就绪表、任务创建和任务调度和切换进行了深入分析。

第四章 μC/OS-Ⅱ实时操作系统的可移植性研究：本章首先提出了移植的概念和 μC/OS-Ⅱ的运行条件。进而介绍了所用的 S3C44B0 开发板，主要研究了 μC/OS-Ⅱ在该开发板的移植。对内核的改造包括：OS_CPU.H 中需要针对具体处理器的字长重新定义一系列数据类型、OS_CPU.H 中需要设置常量来标志堆栈增

长方向、OS_CPU.H 中需要声明用于开关中断和任务切换的宏、OS_CPU_A.ASM 中需要改写 4 个汇编语言的函数、OS_CPU_C.C 中需要用 C 语言写 6 个简单的函数。研究了 μC/OS-Ⅱ初始化和多任务的启动。最后介绍了调试和下载环境，并对移植进行了测试。

第五章 基于 μC/OS-Ⅱ的实时多任务调度算法应用研究：本章研究了单调速率调度算法，对其可调度充分条件和充分必要条件进行了研究。通过和期限最近者优先调度算法进行比较，选择了单调速率调度算法。最后应用 RapidRMA 工具对任务的调度结果进行了分析。

最后对本文所做的工作进行了总结，对运行结果进行了分析，指出了本文的创新之处，并对进一步所要做的研究工作进行了展望。

第二章 实时系统相关理论研究

实时系统通常比较复杂，系统的正确性不仅依赖于计算的逻辑结果而且依赖于结果产生的时间。实时任务要满足处理与时间的关系。它经常要处理很多并发事件的输入数据流，这些事件的到来次序和几率通常是不可预测的，而且还要求系统必须在事先设定好的时限内做出相应的响应。因此，与常见的分时系统(如Windows)相比，实时系统的设计有着更复杂的和不同的考虑因素。实时系统的目标主要是保证系统响应的及时性。

2.1 实时系统

2.1.1 概述

关于实时系统的定义，比较有代表性的是以下两种：

一种是《Oxford Dictionary Of Computer》给出的定义：实时系统是指那些产生输出的时间至关重要的系统。这通常是因为输入对应于外界的某个运动，而输出又必须与同一运动相关。从输入时刻到输出时刻的时间滞后必须充分小，以达到可接受的及时性[4]。这里的“及时性”要在整个系统的背景中加以考虑。例如在导弹制导系统里，输出要求在几个毫秒之内，而在一个计算机控制的汽车装配线上，响应可能只要求在一秒之内。

一种是Jonh.A.Stankovic给出的一个被普遍接受的定义：在实时系统中，计算任务的正确性不仅仅依赖于计算结果逻辑上的正确性，而且还依赖于计算结果产生的时间[5]。

可见，仅仅有正确的输出结果是不够的，输出结果还必须在一个特定的时间间隔内到来。所以，实时系统是与时间约束紧密相关的。通常，实时系统对特定输入做出反应的速度要足以控制发出实时信号的对象；或者，系统能够及时响应外部时间的请求，在规定的时间内完成对该事件的处理，并控制所有的实时设备和实时计算任务协调一致的运行[6]。实时系统的典型应用有：雷达系统中的数据信息处理、飞行器控制系统、交通指挥系统等，这类系统共同特点是需要在规

定的时限内完成任务。

随着计算机硬件的发展，人们对实时系统在性能和安全等方面均提出了更高的要求。根据需求，通常将实时系统分为硬实时系统(Hard Real-Time)和软实时(Soft Real-Time)系统。

硬实时系统中，强制要求任何一个行为动作都要在规定的截止期之前完成，否则将会使整个系统失败。

软实时系统中虽然也有时间限制，但通常情况下偶尔错过几个截止期产生的负面影响不致于引起灾难性的后果，整个系统的功能还是能够实现，只是系统性能有所下降。

2.1.2 特点

除了时间上的约束外，实时系统还具有一些区别于其它系统的特征[7]：

(1)大且复杂。实时系统必须对现实世界的事件做出及时响应，因此必须提供同这些事件关联的多样性。为了应付现实世界中不断变化的需求而重新设计或重新编制软件的成本是高昂的，所以在实时系统的生命周期中要进行不断的维护和增强，并且必须是可扩充的。

(2)可预测性。实时系统对于外部的输入必须是完全可预测的，即使是在最坏的情况下表现出来的行为也应该在系统预先定义的范围之内。

(3)可靠性和容错性。由于被广泛应用于社会生产和生活的关键领域，实时系统对可靠性要求非常高。实时系统运行时产生的错误代价非常高昂，包括产品流失、设备损坏甚至人员伤亡等。因此开发者必须最大限度的保障软件本身的正确性。此外多数实时系统需要被设计成容错的，即在系统的一部分出现错误的情况下尽可能地限制它造成的损失。

(4)并发/并行控制。实时系统经常要对多于一个的外部事件进行响应，而这些外部事件可能是同时发生的，实时系统的设计者可以通过并行和并发的方式来获取更高的系统性能。在多处理器计算机系统中，多个任务能够以真正的并行方式运行，而在单处理器系统中，多个任务通过并发的方式实现逻辑上的“并行”。和并行/并发紧密相关的是调度问题，研究如何为处理器选择特定时间运行的任务，使得实时系统的时间约束得以满足。因此相应的调度算法至今仍然是人们研

究的重点和热点问题。

2.1.3 应用领域

目前实时系统的应用已经渗透到社会生活的各个领域，其典型的应用包括工业控制、军事和生活服务三大方面：

(1)实时系统被广泛应用于各个领域的控制过程，例如电站、数控机床、食品加工车间等。早期的控制系统只包括简单的逻辑控制，因此可以直接由电路实现。而现代复杂的控制系统多数都是由软件主导的，例如一个用来探索未知环境的工业机器人系统就可能是运行在多个异构处理器上的分布式软件系统。

(2)几乎所有的军事软件都是实时系统。它们通常都是开发代价最大、需求最苛刻的软件系统。在这样的系统中实时任务类型多样，系统必须能够为各个任务分别提供不同的、可预测的资源分配。典型的军用实时系统包括武器发射和制导系统、侦察与跟踪系统、部队指挥系统等。

(3)近年来，实时系统越来越多地被应用到人们的日常生活。典型的应用包括交通指挥系统、医疗器械和监护系统、通讯中枢和终端、多媒体流服务和智能家电等。

2.2 实时操作系统

2.2.1 实时操作系统概述

在实时系统中，基本的系统软件是实时操作系统(Real-Time Operating System，RTOS)。它是实时系统应用软件的基础，在许多场合下，实时应用软件都是在RTOS的支持下运行[8]。应用需求的复杂度不断增加，如果实时应用软件的开发还是没有基于一个完善的RTOS，那么无法将系统软件和应用软件分离，每次都要特别指定系统软件和应用软件，这样会造成开发周期过长、成本过高。因此，RTOS的发展是和实时应用需求的复杂化直接相关的，是实时应用软件高效开发的必然要求。实时调度是支持多任务的RTOS的必由功能，各种实时调度算法得到实际应用的前提是结合具体的RTOS设计实现方案，并作为RTOS的一部分实现。

实时操作系统中的关键部分是实时多任务内核，它的基本功能包括任务管

理、定时器管理、存储器管理、资源管理、事件管理、系统管理、消息管理、队列管理等。这些管理功能是通过内核服务函数形式交给用户调用的。使用可占先性内核，所有时间要求苛刻的事件都得到了尽可能快捷、有效的处理。实时操作系统通过有效的服务，如信号量、邮箱、消息队列、延时、超时等使资源得到更好的利用。

从表现上讲，RTOS是实时系统在启动之后运行的一段背景程序。应用程序是运行在这个基础之上的多个任务。RTOS根据各个任务的要求，进行资源(包括存储器、外设等)管理、消息管理、任务调度、异常处理等工作。在RTOS支持的系统中，每个任务都有优先级，RTOS根据各个任务的优先级来动态的切换各个任务，保证对实时性的要求。分时操作系统追求的是吞吐量、公平性，而实时操作系统的目标是最大程度地减少复杂性，保证响应时间，最重要的是它能在规定的时间内可预测地执行任务。实时操作系统中，一个正常情况下需要10个单位时间的任务，由于上下文环境的改变，在最坏的情况下消耗12个的单位时间是可取的。而在普通的操作系统中，一个平均情况下需要3个单位时间的程序经常会执行20个单位时间。

实时操作系统的体系结构包括了硬件抽象层和内核部分。

硬件抽象层(HAL)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RTOS的机器无关代码提供访问和控制服务。

RTOS内核(Kernel)是用来为大多数程序乃至OS(网络、文件系统、驱动程序)构建一系列在抽象的文件上工作的抽象机，使用户程序及上层OS组件对系统设备透明。在提供的RTOS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，方便开发者对不同需求的设备定制驱动程序。

2.2.2 几种常见的实时操作系统

(1)VxWorks实时操作系统。VxWorks是Wind River公司开发的具有工业领导地位的高性能实时操作系统内核。VxWorks的开放式结构和对工业标准的支持，使得开发人员易于设计高效的嵌入式系统，并可以以很小的工作量移植到其它不

同的处理器上。其主要特点包括：快速多任务切换、抢占式的任务调度、任务间通信手段多样化等。且任务间切换时间短、中断延迟小、网络流量大。VxWorks系统具有较好的可剪裁的能力，可剪裁的组件超过80个，用户可以根据自己系统的功能目标通过交叉开发环境方便地进行配置[9]。支持应用程序的动态链接和动态下载，使开发者省去了每次调试都将应用程序和操作系统内核进行链接和下载的步骤，缩短了编辑、调试周期。具有较好的兼容性，使其在不同环境间可以方便的移植，减少了开发周期和经费。

(2)QNX实时操作系统。QNX是加拿大QNX Software system公司的产品。它支持PowerPC、MIPS、ARM等微内核实时平台。QNX是一个实时的、可扩充的操作系统，它部分遵循POSIX相关标准。其内核提供4种服务：任务调度、任务间通信、底层网络通信和中断处理。其任务在独立的地址空间运行，所有其他的服务都实现为协作的用户任务[10]。QNX内核非常小巧而且运行速度极快。用户可以根据实际需要将系统配置成微小的嵌入式操作系统或是包括几百个处理器的超级虚拟机操作系统。

(3)Windows CE操作系统。Microsoft Windows CE是一个32位的多任务嵌入式操作系统，它的模块化设计允许用户方便地进行定制，它是微软“维纳斯计划”的核心，包含了Internet Explorer，可以和Internet实现连接、同步。系统的基本内核需要至少200K的RAM。从SEGA的DreamCast游戏机到现在的大部分高端掌上电脑都采用了Windows CE，但由于其价格较高，使整个产品的成本急剧上升[11]。

(4)μC/OS-II操作系统。μC/OS-II是由美国人Jean J.Labrosse编写的一个完全免费的嵌入式实时操作系统。μC/OS-II是一个源代码开放、结构小巧、具有可剥夺实时内核的实时操作系统。其内核提供任务调度与管理、时间管理、任务间同步与通信、内存管理和中断服务等功能。μC/OS-II适合小型控制系统，具有执行效率高、占用空间小、可裁剪、实时性能优良和可扩展性强等特点[12]。μC/OS-II内核是针对实时系统的要求设计实现的，相对简单，可以满足较高的实时性要求，并且针对各种处理器的移植相对简单。

上述介绍的Windows CE、VxWorks、QNX实时操作系统都是商业软件，价格非常昂贵，不公开源代码，这些都极大的限制了国内许多开发商的开发使用。而且在嵌入式系统的开发中，不公开系统源代码给上层应用开发人员带来了极大的不便。虽然Linux源代码公开，但是Linux不支持任务抢占，所以在实时性方面

不能称为真正的实时内核。

由Jean J.Labrosse开发的嵌入式实时操作系统μC/OS-II就弥补了上述缺点，它的源代码公开，为广大嵌入式爱好者和开发者提供了学习、研究嵌入式操作系统的一个优秀范例，它的内核是占先式内核设计，属于真正的实时系统。通过对该系统的学习和研究，可以了解一个实时操作系统在系统内核结构、任务管理、任务间通讯和同步等方面的实现方法和一些设计技巧。通过对该系统源代码的深入研究，可以自主开发出满足开发项目需要的操作系统内核，而且针对该系统中的一些不足之处，也可以在实时性和稳定性方面加以改进，可以在具体的项目中针对不同主芯片的特点做出最大的系统优化。因此本文的研究工作是基于μC/OS-II操作系统。

2.2.3 实时操作系统的衡量标准

用来作为RTOS性能衡量标准的指标主要有如下几种：

(1)系统响应时间(System Response Time)：系统在发出处理要求到系统给出应答信号的时间；

(2)任务切换时间(Context-Switching Time)：多任务之间进行切换而花费的时间；

(3)中断延迟时间(Interrupt Latency Time)：从接收到中断信号到操作系统做出响应，并完成进入中断服务程序的时间。

2.2.4 实时操作系统的实时内核

在多任务系统中，内核负责管理各个任务，或者说为每个任务分配CPU时间，并且负责任务之间的通信。内核提供的基本服务是任务切换。使用实时内核之所以可以大大简化应用系统的设计，是因为实时内核允许将应用分成若干个任务，由实时内核来管理它们。内核本身也增加了应用程序的额外负荷，代码空间增加ROM的用量，内核本身的数据结构增加了RAM的用量。而且更加主要的是，每个任务要有自己的栈空间，这是相当可观的容量。内核本身对CPU的占用时间一般在2到5个百分点之间。实时内核按CPU控制权的分配方法的不同，通常分为非抢占式内核和抢占式内核。

2.2.4.1 非抢占式内核

非抢占式内核要求每个任务自我放弃CPU的所有权。非抢占式调度法也称作合作型多任务，各个任务共享一个CPU。异步事件还是由中断服务来处理，中断服务可以使一个高优先级的任务由挂起状态变为就绪状态。但中断服务完成以后，控制权还是回到原来被中断的那个任务，直到该任务主动放弃CPU的使用权。此时那个高优先级的任务才能获得CPU的使用权。

非抢占式内核的一个优点是响应中断快。在任务级，非抢占式内核允许使用不可重入函数。每个任务都可以调用不可重入性函数，而不必担心其它任务可能正在使用该函数，从而造成数据的破坏。因为每个任务要运行到完成时才释放CPU的控制权。当然该不可重入型函数本身不得有放弃CPU控制权的企图。使用非抢占式内核时，任务级响应时间比前后台系统快得多。此时的任务级响应时间取决于最长的任务执行时间。

非抢占式内核的另一个优点是，几乎不需要使用信号量来保护共享数据。运行着的任务占有CPU而不必担心被别的任务抢占。但在某种情况下，信号量还是用得到，如在处理共享I/0(Input/Output)设备时仍需要使用互斥型信号量。非抢占式内核如图2-1所示。

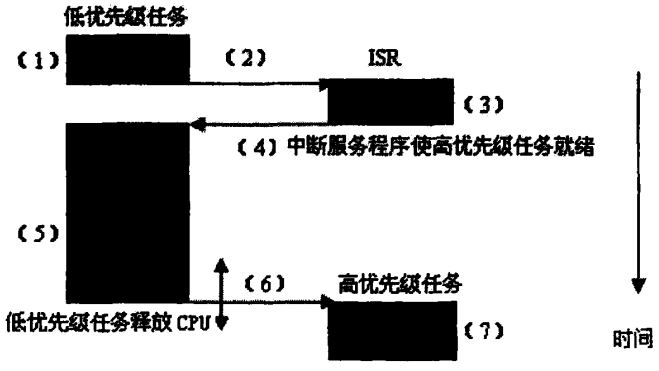

图2-1 非抢占式内核

Fig. 2-1 Non Preemptive Kenel

任务在运行过程之中，如果中断是开着的，[F2-1(1)]中断来了，CPU由中断向量[F2-1(2)]进入中断服务子程序做事件处理[F2-1(3)]，使一个优先级更高的任务进入就绪态。中断服务完成以后，中断返回指令[F2-1(4)]使CPU回到原来被中断的任务，接着执行该任务的代码[F2-1(5)]直到该任务完成，调用一个内核服务函数以释放CPU控制权，由内核将控制权交给优先级更高的、并已进入就绪态的

任务[F2-1(6)]，这个更高优先级任务才开始处理中断服务程序的事件[F2-1 (7)]。

非抢占式内核的最大缺陷在于其响应时间。高优先级的任务已经进入就绪态，但还不能运行，也许要等很长时间，直到当前运行着的任务释放CPU。非抢占式内核的任务级响应时间是不确定的，最高优先级的任务何时能够拿到CPU控制权，完全取决于应用程序什么时候释放CPU。

总之，非抢占式内核允许每个任务运行，直到该任务自愿放弃CPU的控制权。中断可以打断运行着的任务，中断完成后将CPU控制权还给被中断任务。任务级响应时间好于前后系统，但仍是不可知，目前商业软件中几乎没有非抢占式内核。

2.2.4.2 抢占式内核

如果系统对响应时间要求很高，需要使用抢占式内核。μC/OS-Ⅱ以及绝大多数商业上销售的实时内核都是抢占式内核。最高优先级的任务一旦就绪，总能得到CPU的控制权。当一个运行着的任务使一个比它优先级高的任务进入了就绪态，当前任务的CPU使用权就被剥夺了，或者说被挂起了，那个高优先级的任务立刻得到了CPU的控制权。如果是中断服务子程序使一个高优先级的任务进入就绪态，中断完成时，中断了的任务被挂起，优先级最高的那个任务开始运行。抢占式内核如图2-2为所示。

使用抢占式内核，最高优先级的任务何时可以得到CPU的控制权是可预知的。使用抢占式内核使得任务级响应时间得以最优化。抢占式内核总是让就绪态的高优先级任务先运行，中断服务程序可以抢占CPU，到中断服务完成时，内核让此时优先级最高的任务运行(不一定是那个被中断了的任务)。任务级系统响应时间得到了最优化，而且是可知的。

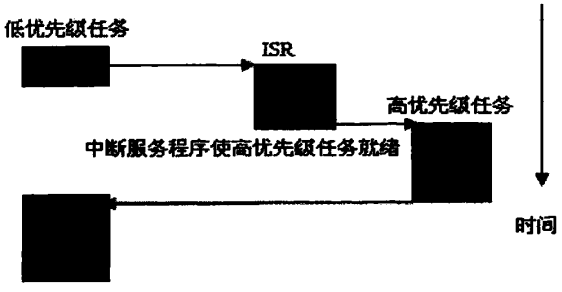

图2-2 抢占式内核

Fig. 2-2 Preemptive Kenel

2.2.4.3 实时内核的特点

1）可抢占性。由于涉及到中断处理、可重入代码等问题，许多操作系统内

核都存在一些禁止中断或不可抢占的代码段(如：Linux、UNIX、Windows)。不论哪种情况都可能造成实时任务无法在预定的时刻开始执行，或者错过截止期。因此，实时内核应具有尽可能少的不可抢占代码，并且其不可抢占代码的执行时间应当尽可能短，最好是确定的和可预测的。

2）多任务并发。由于一个系统常需要同时支持多个应用同时进行，一个应用又往往由一组任务完成，所以能够并发运行多个任务是很重要的。多任务提供了一个较好的对现实世界的匹配，因为它允许对应于许多外部事件的多线程执行。另外系统内核对多任务并发的运行还有利于更充分的利用系统资源。

3）实时调度。为使并发运行的多个实时任务能够及时地分配到系统资源，在规定时间之前完成运算，需要一套严密的实时调度方案为它们安排执行顺序。不论按什么样的调度方法进行调度，其目的都是即要并发运行，又能保证各个任务的时间约束。

4）快速灵活的任务间通信与同步。在一个实时系统中，可能有许多任务作为一个应用的一部分执行，它们之间具有协作关系。系统必须提供这些任务间的快速灵活且功能强大的通信机制。同时也要提供为了有效共享不可抢占的资源或临界区所需的同步机制。

5）合理的中断机制。中断是计算机硬件与操作系统之间的纽带，也是RTOS的基础部分之一，能够在确定的响应时间内对中断进行及时处理是RTOS的必要功能。当多个中断在非常接近的时刻到来时，中断机制一方面要保证不丢失中断，另一方面还要尽可能的减少中断延迟。因此，需要有合理的中断机制的设计。

综合分析以上各项基本要求，可以发现实时调度与其它部分有着不同程度的相关性。例如，实时内核的可抢占性决定了实时调度策略是否可采用抢占式调度；多任务并发是实时调度实现的基本前提；中断最大延迟是影响实时调度精度的因素之一等。实时调度在实时操作系统中的实现将涉及到实时内核的许多方面。

2.3 实时系统中的调度

由实时系统的定义和特征可知，实时系统与非实时系统的最大不同就是要满足处理和时间的关系。特别是在多任务并发的系统中，如要确保各个实时任务的时间约束都能得到满足，就要对并发运行的任务集合在系统处理机上的执行进行合理的安排。这样才能达到充分利用系统资源完成实时计算的目的，于是便涉及

实时系统研究的一个关键问题——调度问题。

调度的实质是一种资源的分配，即如何在系统的各个任务间合理的分配运行所需的资源。调度算法决定系统如何进行资源分配，它是一种服务于系统目标的策略。对于不同的系统及系统目标，应设计不同的调度算法。常见的通用操作系统(General Purpose Operating System)，如：Windows、UNIX、Linux等，其中的调度算法所追求的设计目标在于优化系统的平均性能[13]。这样的设计目标适合于一般的桌面计算机应用和某些网络服务器应用环境，而不适合解决实时系统中的问题。

实时调度面临着与原来的通用操作系统调度所不同的考虑因素，有效的保障系统中的每个实时任务的实时性是实时调度的一个重要目标，这里的实时性能是指响应时间、截止期错过率等与时间相关的指标。由于实时调度是保障实时系统特性的重要手段，因此，实时调度历来是实时系统研究中的关键问题。

2.3.1 实时调度基本理论

2.3.1.1 基本概念

1）进程：处于执行期的程序以及它所包含的资源的总称。进程并不仅仅局限于一段可执行的代码，通常还包含其它资源，如用来存放全局变量的数据段、打开的文件、挂起的信号等[14]；

2）任务：完成某一特定功能的软件实体，是实时调度中的一个基本单位；

3）作业：任务在其生命周期中的某一次执行称为该任务的一个作业；

4）实时任务：具有实时性能需求的任务称为实时任务，反之则称为非实时任务；

5）实时应用：指具有实时性能需求的应用程序，它往往是由一组相关的实时任务组成；

6）就绪态：当进程已分配到处理机以外的所有必要资源的时候，它能够被系统调度执行，此时的状态称为就绪态；

7）执行态：指进程已获得处理机，其程序正在执行；

8）阻塞态：进程因发生某种事件而暂停执行时的状态，也称等待状态或睡眠状态。

2.3.1.2 实时任务的分类

从不同的角度对实时任务加以分类：

1．根据任务的作业到达时刻规律的不同，可分为周期任务(Periodic Task)、间发任务(Sporadic Task)和非周期任务(Aperiodic Task)。

1)周期任务：相邻两次作业到达时刻间的间隔是一个固定的常数(即周期)；

2)间发任务：相邻两次作业到达时刻之间的间隔不固定，但有一个下限值；

3)非周期任务：它的作业到达时刻没有规律。

2．根据对截止时间的要求(即实时性能要求的程度)不同，可分为硬实时任务和软实时任务。

1)硬实时任务(Hard Real-Time Task)：系统必须满足任务对截止时间的要求，在某个限定时刻之前不能完成任务将出现难以预测的结果；

2)软实时任务(Soft Real-Time Task)：它也受到截止时间的约束，但并不严格，若偶尔错过了任务的截止时间，对系统产生的影响不会太大。

2.3.1.3 常见的任务模型

在描述具体的任务模型之前，首先给出下面讨论中将涉及到的任务及其有关参数的定义与表示。对于任务τ_i具有下面的参数[15]：

a_i表示任务的到达时间，即任务被启动并准备执行的时间(也就是说，任务由其它状态转变为就绪态的那个时刻)。

C_i表示任务的最坏情况执行时间(Worst Case Execution Time，WCET)，即任务在最坏情况下无中断执行所需的处理器时间。WCET常作为实时调度中任务对处理机的可能占用情况的一个衡量值。

c_i表示任务的剩余执行时间，即任务的最坏情况执行时间与任务已经执行时间的差。

d_i表示任务的绝对截止期，即任务在这个时间应该完成执行并产生一个有价值的结果。

D_i表示任务的相对截止期，有$D_i=d_i-a_i$。

T表示任务的周期。

Φ_i表示相位偏移，即第一个作业的到达时刻相对于某个时间原点的差。

P_i表示任务的优先级。

截止期错过率(Deadline Missing Ration，DMR)：假设在某一段确定的时间内，

错过其截止期的作业数为n，正常完成的作业数为m，则DMR=$\frac{n}{m+n}$。DMR是描述实时任务在实际运行中，其实时性需求满足情况的一个重要衡量值。

设含有n个周期任务的周期任务集$S=\{\tau_1,\tau_2,\ldots,\tau_n\}$，相应的任务周期分别是$T_1, T_2,\ldots, T_n$，每个周期内的最坏情况执行时间分别为$C_1, C_2, \ldots, C_n$，相对截止期为$D_1$，$D_2$，…，$D_n$，对于第$i$个周期任务，必须满足$C_i \leqslant D_i \leqslant T_i$。

处理机利用率表示任务集对处理机资源的占用程度，周期任务集的处理机利用率可以用公式(2-1)计算[16]：

$$U_s=\sum_{i=1}^{n}\frac{C_i}{T_i} \tag{2.1}$$

当处理机特指计算机的CPU时，处理机利用率也称为CPU利用率。

在实时调度算法的研究中，常对上述模型给出一些限制条件：

(1)$D_i \leqslant T_i (1 \leqslant i \leqslant n)$；

(2)任务可以在任意时刻被抢占；

(3)忽略所有系统开销，如任务抢占、上下文切换时的开销等；

(4)周期任务的每个新作业在周期开始之后立即到达；

(5)任务的作业间相互独立；

为便于进行算法分析，有些场合还会加上这样两个更严格的假设限制：

(1)$D_i=T_i$（$1 \leqslant i \leqslant n$）

(2) $\Phi_i=0$（$1 \leqslant i \leqslant n$）

目前对周期任务的调度研究已取得了丰富的成果，常用的是基于优先级策略的方法，最典型就是速率单调(RM)算法和最早截止期优先(EDF)算法。这两种算法及其扩展方法也被深入研究，并且至今一直被广泛应用。

为了使周期任务模型也适用于间发任务和非周期任务，T被赋予不同的含义就可以了。对于间发任务，T被解释为最小到达时间间隔(Minimum Inter-arrival Interval)，例如T为20ms的一个间发任务保证每20ms到达的次数不会超过一次；对于非周期任务，T可以理解为平均到达时间间隔。

2.3.2 调度研究现状

实时调度是保障实时系统重要特性的重要手段之一，国内外在这方面的研究

也由来已久。因为实时调度策略的设计与任务及系统的特性有关，因此对调度方法的研究也多种多样。调度算法即用以确定所要调度的任务能够在某一特定时刻执行的规则集。

2.3.2.1 针对单一任务类型的调度

1．基于优先级的调度策略(Priority-Driven Scheduling, PD)

基于优先级的调度算法给每个进程分配一个优先级，在每次进程调度时，总是调度那个具有最高优先级的任务来执行。根据不同的优先级分配方法，基于优先级的调度算法可以分为静态优先级调度算法和动态优先级调度算法。对于一种调度算法，如果任务的优先级可以根据任务请求改变，称其为动态调度算法。动态调度算法是在运行期间才决定选择那个就绪任务来运行，它所根据的是目前已经处于就绪态的各个任务的相关属性（如期限、松弛时间），以此来决定当前的调度序列。如果任务的优先级是事先确定好的，称其为静态优先级。静态调度算法是在任务集在运行之前就产生一个静态的调度表，在运行时总是按照调度表决定从就绪任务列队中选择哪个任务来运行，这类调度算法假设系统中的实时任务特性（如周期、执行时间、期限等）是已知的，可调度性分析是脱机进行的。

- 静态优先级调度算法有固定优先级调度算法、时钟驱动调度算法和RM算法。

固定优先级调度算法广泛应用于实时系统和内核中，每个任务在运行之前都已经分配好固定和静态的优先级，任务的调度顺序取决于分配给它们的优先级。在系统过载时，固定优先级调度算法体现出了很好的可预测性[17]。

时钟驱动调度算法是指在系统开始执行之前，选择一些待定的时刻，在这些时刻决定哪一个作业在何时执行。在一个使用时钟驱动调度方法的系统里，所有强实时作业的参数都是固定并且已知的。作业的调度表脱机计算被保存(也称静态调度)，然后在运行时使用。依据此调度，调度程序在每一个调度决策时刻调度作业运行[18]。用这种方法，运行时的调度开销可被最小化。通常是在有规律的空白时刻做出调度决策。

RM算法是由Liu于1973年提出的。它根据任务执行周期的长短来决定调度优先级，执行周期小的任务具有较高的优先级。由于RM算法已被证明是静态优先级调度算法中最优的，所以它受到研究者和应用开发者的重视。为了提高RM算法的处理机利用率，人们提出了若干改进方法，典型的有：子任务法、双优先级

法和最优双优先级法[19,20]。这些算法相对于RM算法略有提高，但都是以复杂度和系统开销为代价。

● 动态优先级调度算法有最早截止期优先EDF (Earliest Deadline First)算法和最小松弛时间算法(Least Slack Time First，LST)。

EDF算法是由Layland提出的。EDF算法是动态优先级策略中的最优算法，对整个任务的调度开销进行全面比较，选择最高优先级任务进行调度，有较小的调度开销。

LST算法计算任务的松弛时间，即任务距离其截止期的时间与它还需要执行的时间之差。在任意时刻把最高优先级分配给具有最小松弛时间的任务。它也被证明是最优动态优先级算法。

2．基于比例共享的调度策略(Share-Driven Scheduling，SD)

比例共享调度算法指基于CPU使用比例的共享式的调度算法，其基本思想就是按照一定的权重(比例)对一组需要调度的任务进行调度，让它们的执行时间与它们的权重完全成正比。属于这一类的算法目前已有多种方案被提出，公平共享GPS(Generalized Processor Sharing)算法即是较早被提出的一种[21]。此外最早完成时间优先的共享处理器EGPS(Earliest-Completion-Time GPS)算法也是一种较典型的算法，它对GPS算法进行了改进，将其运用到实际系统中[22]。

3．基于时间表的调度策略(Time-Driven Scheduling，TD)

对于那些具有稳定、已知输入的简单系统，可以使用时间驱动的调度算法，它能够为数据处理提供很好的预测性。这种调度算法本质上是一种设计时就确定下来的离线的静态调度方法。在系统的设计阶段，在明确系统中所有的处理情况下，对于各个任务的开始、切换、以及结束时间等就事先做出明确的安排和设计。这种调度算法适合于那些很小的嵌入式系统、自控系统、传感器等应用环境。

2.3.2.2 针对各种类型实时任务的混合调度

在实际的实时应用中，常常会遇到周期任务和非周期任务同时存在，硬实时与软实时需求同时存在的情况。例如，在实时监控系统中，数据采集任务是硬实时周期任务，与用户交互的任务是非周期软实时任务。任务的时间性约束需求不同，使得系统在调度处理这些任务时的策略与目标也是不同的。在针对各种类型实时任务的混合调度方面，已经有一些研究成果。以下是几种典型方法：

1)后台处理(Background Processing)法[23]。这种方法首先满足前台周期任务的

运行需要，在处理器空闲(未执行任何周期任务并且没有未完成的周期任务)时以先来先服务FCFS(First Come First Server)次序调度执行后台非周期任务。

2)基于服务器(Server Based)方法。又称为带宽预留(Bandwidth Preserving)算法。其主要思想是，在保证满足周期任务截止期的前提下，引入一个或者几个额外的周期任务，使用指定的处理器带宽作为服务器来处理非周期任务，使非周期任务尽可能快地执行[24]。根据周期任务调度算法的不同，带宽预留算法又可以归结为固定优先级服务器算法与动态优先级服务器算法，典型的是由Lehoczky等提出的可延期服务器DS(Deferrable Sever)算法和限定截止期的可延期服务器DDS(Deadline Deferrable Sever)算法等[25]。

- 固定优先级服务器算法假定周期任务由单调速率(RM)算法调度。它可以分为轮询服务器PS(Polling Server)方法、可延期服务器DS(Deferrable Server)方法和偶发服务器SS(Sporadic Server)方法[26,27]。

轮询服务器算法是一种最简单的固定优先级服务器算法，其基本思想是建立一个具有固定优先级（通常为最高优先级）的轮询任务以固定的周期来处理任何未完成的非周期任务，如果当前系统中没有非周期任务，则这个轮询任务挂起直到下一个周期，原本分配给非周期任务的处理时间则被放弃。尽管轮询算法比后台处理方法有一定的性能改进，但是仍存在很大的改进余地。

可延期服务器算法是由Lehoczky等提出的，其基本思想也是使用一个特定的高优先级周期任务为非周期任务提供服务，但是不同于轮询算法之处在于，如果当启用这个服务任务时没有挂起的非周期任务，则采用一定的策略为非周期任务保持这些分配的执行时间，以便当非周期任务到达时能够及时地处理。

偶发服务器算法是由Sprunt等提出的一种更好的服务机制，具有简单而且高效的双重优点。与DS算法相比，SS算法可实现更高的处理器利用率，并且在较高的非周期负载下能够提供比DS算法更好的平均响应时间。

- 动态优先级服务器算法是基于EDF动态调度策略基础上提出的一系列算法。限定截止期的可延期服务器DDS(Deadline Deferrable Server)算法和限定截止期的偶发服务器DSS(Deadline Sporsdic Server)是由Ghazalie提出的DS算法和SS算法的动态扩展算法。DDS算法的服务器采用一个相对截止期等于周期T的周期任务，在每个周期的开始，它被分配一个固定大小的预执行时间，如果所有等待的非周期任务都被处理完，而预算执行时间并未耗尽，则这些执行时间被保持直

到周期结束，并且能够用来处理在这段时间内到达的非周期任务；如果预执行时间耗尽，则DDS服务器将被挂起直到下一个周期。DSS算法保持了SS算法的基本思想，而修改之处在于通过选择合适的截止期为服务器分配优先级。但是DSS算法需要跟踪任意数量的、具有不同长度与补充时间的执行时间块，尤其当同时要求多个具有不同能力与优先级的服务器时，DSS算法的实现很复杂。

完全带宽服务器TBS（Total Bandwidth Server）算法是Spuri和Buttazzd提出的一种全新的带宽预留算法。其主要思想是为每个非周期任务请求分配一个尽可能早的截止期，从而解决如果服务器具有较长的周期而导致非周期任务的执行可能被很大程度地延迟的问题。此外，稳定带宽服务器CBS(Constant Bandwidth Server)算法与TBS算法相似，只是截止期的计算方法和服务器容量补充时机不同而已[28]。虽然这两种方法最初都是针对非周期任务提出的，但是基于服务器策略的实时调度可以为系统中的每个实时应用提供一个具有确定带宽的虚拟处理机，并且具有易扩展、易配置等特点。

3)基于时间空闲(Slack Based)方法。该算法的主要思想是通过离线或者在线分析从周期任务调度的空隙获得尽可能多的处理时间来处理非周期任务。基于空闲时间的方法主要包括空闲时间偷取(Slack Stealing)法、时间片移位(Slot Shifting)法与双重优先级(Dual Priority)法[29-32]。

空闲时间偷取算法SSA(Slack Stealing Algorithm)是基于空闲时间(Slack Time)的概念，时间段的空闲时间等于能够用来执行非周期任务而不会促使任何周期任务错失截止期的时间总数。从某种意义上来说SSA算法是一种贪婪算法，一旦有可用的空闲时间就会立即被使用，这种算法是各种贪婪算法中最优的算法。

时间片移位(Slot Shifting)算法是Fohler针对静态调度系统中的混合任务集的联合调度问题提出的，这个算法首先要为周期任务构造一个静态调度表，满足任务的时间、同步与通信需求，接着离线计算空闲处理时间的大小与分布，记录每个空闲处理时间的位置以及随后的周期任务执行允许的偏移时间。由于所有未用的处理时间的数量与分布是离线确定的，只需在线地增量维护，因此运行机制相对简单并且要求较少的内存。

双重优先级(Dual priority)算法。双重优先级方法是一种简单的利用空闲时间调度处理软实时任务的算法。双重优先级调度最早由Dabis等提出，算法中定义高、中、低3个优先级层次，每个硬实时任务被分配两个优先级，一个在高优先

级层次，另一个在低优先级层次，通常任务之间的优先级顺序在这两个优先级层次上保持一致，而非周期任务运行在中优先级层次上。

在国内，也有一些高校和科研机构对实时系统的调度进行了广泛研究，但大多数是针对某种调度算法进行改进和完善，例如，文献[33-36]等。

随着实时系统应用的日益广泛，多类型实时任务并存的情况给系统调度方法带来了新的需求和挑战。由于操作系统调度所面对的任务种类的多样性，以及任务执行环境的动态变化的不可预测性，使得系统中要处理的任务的类型、到达时刻、所需计算资源数量等通常是不可预见的，在这种动态实时系统中，需要灵活的可扩展性和动态配置能力。而目前的技术还不能完全满足实际需求，而且有关支持各种类型任务共存的具有复杂实时应用特性的调度方法的研究还很欠缺，因此，有待进一步研究与探索。

2.4 本章小结

本章首先介绍了实时系统的概念、特点及应用领域。然后介绍了 Linux、VxWorks、QNX、WindowsCE、μC/OS-II 等实时操作系统，通过对比选定了本文所采用的操作系统——μC/OS-II 操作系统。介绍了实时操作系统的概念、体系结构、衡量标准和实时内核，分析了实时内核的特点，并以图形的形式剖析了抢占式内核和非抢占式内核的工作原理。最后研究了实时系统中的调度问题。包括调度的概念、实时任务的分类及常见的任务模型；从针对单一任务调度算法和各种类型任务的混合调度算法，对调度现状进行了分析研究。

第三章 μC/OS-II实时操作系统研究

3.1 概述

μC/OS-II是一个完整的、可移植、可裁剪、稳定可靠的占先式实时嵌入式多任务内核，具有任务调度、任务管理、时间管理、内存管理和任务间的通信与同步等基本功能。其包含全部功能的核心部分代码只占用8.3K字节，由于其可裁剪性，用户系统中实际的代码最少可达2.7K字节。

μC/OS-II支持多任务管理，最多可达64个任务。通过信号量、邮箱和消息队列实现任务间的通信。μC/OS-II具有中断管理功能，支持多达255层的中断嵌套。它的大部分是用标准ANSI C编写的，还有一小部分汇编语言代码，结构简单，可移植性好。这些特性使得它支持不同架构的微处理器。目前，μC/OS-II已在超过40种不同架构的微处理器上运行，包括从8位到64位的微处理器、微控制器和数字信号处理器。和μClinux相比，μC/OS-II具有很强的实时性，和Windows CE相比，μC/OS-II有源代码开放、学习用免费、商用费用低的优势，因此在实时性要求高的控制领域，μC/OS-II是一个不错的选择。

μC/OS-II的任务调度是按抢占式多任务系统设计的，即它总是执行处于就绪条件下优先级最高的任务。μC/OS-II规定所有任务的优先级必须不同，任务的优先级唯一的标识了该任务。高优先级的任务在处理完后必须进入等待或挂起状态，否则低优先级的任务永远也不可能执行。系统通过两种方法进行任务调度：一是时钟节拍或其它硬件中断到来后系统会调用函数OSIntCtxSw()执行切换功能；二是任务主动进入挂起或等待状态，这时系统通过发软中断命令或依靠处理器执行陷阱指令来完成任务切换，中断服务例程或陷阱处理程序的向量地址必须指向函数OSCtxSw()。

对于一个多任务操作系统，任务间的通信与同步是必不可少的。μC/OS-II提供了四种同步对象，分别是信号量、邮箱、消息队列和事件。所有的同步对象都有相应的创建、等待、发送的函数。但这些对象一旦创建就不能删除，所以要避免创建过多的同步对象以节约系统资源，也可以根据系统的 实际需要选择使用

合适的同步对象。

μC/OS-Ⅱ的优点还在于它的源代码开放、结构简单紧凑、脉络清晰，便于初学者学习和使用。可以在短时间内掌握它的结构和工作原理，然后在它上面开发应用程序。虽然μC/OS-Ⅱ具有上述优点，但它也存在不足，它只是一个核，本身没有文件系统和网络通信功能，如果需要实现这些功能，需要对μC/OS-Ⅱ进行扩展。

3.2 μC/OS-Ⅱ结构

μC/OS-Ⅱ的硬件软件体系结构如图3-1所示。

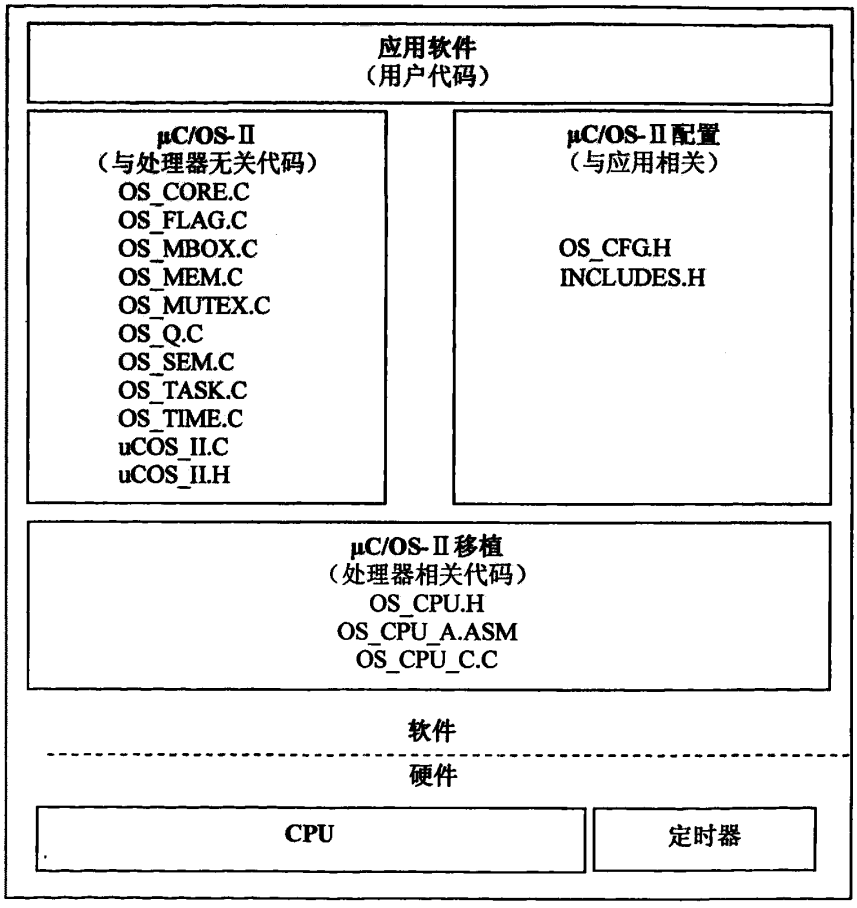

图3-1 μC/OS-Ⅱ的硬件/软件体系结构

Fig.3-1 Hardware/Software System Structure of μC/OS-Ⅱ

核心部分(OSCORE.C)是操作系统的处理核心，包括操作系统初始化、操作系统运行、中断进出的前导、时钟节拍、任务调度、事件处理等部分。核心部分模块包含了维持系统运作的基本支持功能。

μC/OS-Ⅱ是以任务为基本调度单位，任务处理部分(OSTASK.C)中的内容都是与任务的操作密切相关的。包括任务的建立、删除、挂起、恢复等等。任务同步和通信部分为事件处理部分，包括信号量、邮箱、邮箱队列；以及事件标志等部分，主要用于任务间的互相联系和对临界资源的访问。

μC/OS-Ⅱ中的最小时钟单位是time tick(时钟节拍)，而任务延时等操作是由时钟部分(OSTIME.C)完成的。

μC/OS-Ⅱ的内存管理不同于通常意义上的内存管理，μC/OS-Ⅱ的内存管理主要是面向用户任务中的动态内存分配。

移植相关部分是指μC/OS-Ⅱ对所使用的CPU平台的移植部分。由于μC/OS-Ⅱ是一个通用性的操作系统，所以对于关键问题上的实现，还是需要根据CPU的具体内容和要求作相应的移植。这部分内容通常用汇编语言编写。主要包括中断级任务切换的底层实现、时钟节拍的产生和处理、中断的相关处理部分等内容。

3.3 μC/OS-Ⅱ内核特点

μC/OS-Ⅱ实际上只是一个实时操作系统内核，只包含了任务调度、任务管理、时间管理、内存管理和任务间的通信与同步等基本功能，没有提供输入输出管理、文件系统、网络之类的额外服务。但是μC/OS-Ⅱ为嵌入式开发者提供了健壮的、可靠的程序开发环境，μC/OS-Ⅱ有以下几个主要特点[37]：

(1)源码公开：很容易在网上找到μC/OS-Ⅱ的全部源码，且注解详细，组织有序，还有专门解释代码的工作原理与组织这些代码的资料。

(2)可裁剪性：可以只使用μC/OS-Ⅱ中应用程序需要的那些系统服务。用户可根据需要选择适当模块来裁剪和配置系统，这样，通过目标模块之间的按需组合，可以减少产品中的μC/OS-Ⅱ所需的存储空间，这种裁减性是依靠条件编译实现的。另外，由于μC/OS-Ⅱ源码的开放特性，用户还可以针对自己的硬件优化代码，以获得更好的性能。

(3)可移植性：μC/OS-Ⅱ的源码绝大部分是用移植性很强的ANSIC编写的。和微处理器硬件相关的那部分是用汇编语言写的。汇编语言写的部分已压到最低限度，使得μC/OS-Ⅱ便于移植到其它微处理器上，只需要该处理器有堆栈指针，有CPU内部寄存器入栈、出栈指令。另外，使用的C编译器必须支持内嵌汇编或

者该C语言可扩展、可连接汇编模块，使得关中断、开中断能在C语言程序中实现。目前μC/OS-Ⅱ可以在绝大多数8位、16位、32位和64位微处理器、微控制器、数字信号处理器上运行。

(4)可剥夺性：μC/OS-Ⅱ是完全可剥夺型的实时内核，总是运行就绪条件下优先级最高的任务，大多数商业内核也是可剥夺型的。

(5)多任务：μC/OS-Ⅱ可以管理64个任务，目前保留8个给系统。应用程序最多可以有56个任务。每个任务被赋予的优先级不同。

(6)中断管理：系统可通过中断使正在执行的任务暂时挂起。如果优先级更高的任务被该中断唤醒，则高优先级的任务在中断嵌套全部退出后立即执行，中断嵌套层数可达255层。

(7)可固化：μC/OS-Ⅱ是为嵌入式应用而设计的，只要具备合适的系列软件工具(C编译、汇编、链接及下载/固化)，μC/OS-Ⅱ就可以嵌入到产品中成为其一部分。

(8)可确定性：全部μC/OS-Ⅱ的函数调用与服务的执行时间具有可确定性。μC/OS-Ⅱ系统服务的执行时间不依赖于任务的多少。

(9)任务堆栈：μC/OS-Ⅱ中每个任务有自己单独的堆栈。μC/OS-Ⅱ允许每个任务有不同的堆栈空间，以便压低应用程序对RAM的需求。

(10)系统服务：μC/OS-Ⅱ提供很多系统服务(例如邮箱、消息队列、信号量、块大小固定的内存申请与释放、时间相关函数等)。

3.4 μC/OS-Ⅱ内核的任务管理

所谓任务是指一个具有独立功能的无限循环的程序段的一次运行活动，是实时内核调度的单位，具有动态性、并发性、异步独立性[38]，多表现为一个独立的线程。任务包含可执行代码、数据、堆栈和程序执行的上下文环境等内容。按照任务到达情况的可预测性，可分为周期性和非周期性任务。任务的特性可以通过优先级(Priority)、周期(Period)、计算时间(Computation Time)、就绪时间(Ready Time)、期限(Deadline)等参数来进行描述。对一个任务要进行可调度性分析，至少要指定其周期、计算时间、期限这三个参数。一个系统是否可调度，就是该系统任务集中的每个任务能否在各自的期限内完成。μC/OS-Ⅱ内核的灵巧设计在任务调度方面得到集中的体现。

任务管理是μC/OS-II内核提供的最主要的功能之一。任务管理方面，μC/OS-II最多可以管理64个任务，这些任务通常都是一个无限循环的函数。在目前的版本中，保留了优先级为0、1、2、3、OS_LOWEST PIRO_3、OS_LOWEST PIRO_2、OS_LOWEST PIRO_1、OS_LOWEST PIRO_0的任务，所以用户可以同时拥有56个任务。其中，0表示最高优先级，OS_LOWEST PIRO_63表示最低优先级。μC/OS-II提供了任务管理的各种函数调用，包括创建任务、删除任务、改变任务的优先级、挂起和恢复任务等。系统初始化时会自动产生两个任务：一是空闲任务OSTaskIdle，它的优先级最低为OS_LOWEST PIRO，该任务只是不停地给一个32位的整型变量加1；另一个是统计任务OSTaskStat()，优先级为OS_LOWEST PIRO_1，该任务每秒运行一次，负责计算当前CPU的利用率。

3.4.1 用户任务结构

一个任务通常是一个无限的循环。一个任务看起来像其它C的函数一样，有函数返回类型，有形式参数变量，但是任务是绝不会返回的。μC/OS-II中任务的结构如下代码所示：

```
void Task(void P*data){
for(;;){
/*用户代码*/
CallμC/OS-II services
OSTimeDly();//释放资源
/*用户代码*/
}
}
```

任务完成以后，可以自我删除，但任务代码并非真的删除，μC/OS-II只是简单地不再理会这个任务，这个任务的代码也不会再运行，如果任务调用了OSTaskDel()，这个任务不会返回什么。删除任务的伪代码如下：

```
void Task(void P*data){
/*用户代码*/
OSTaskDel(this TaskPriority);//使任务所占有的优先级位置为空
}
```

3.4.2 任务的状态

图3-2是μC/OS-Ⅱ的任务状态转换图。在任一给定时刻，任务的状态一定是这五种状态之一。

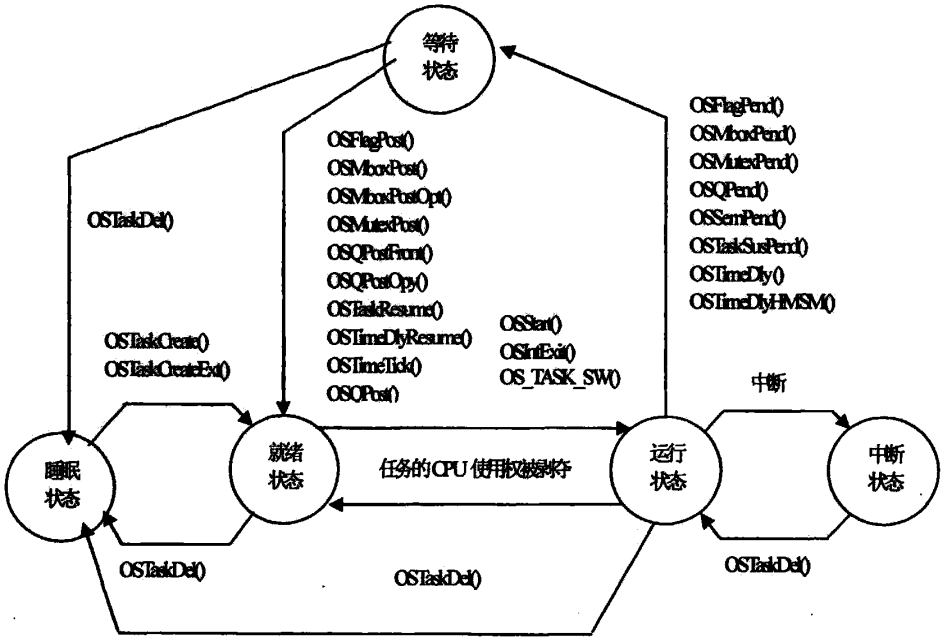

图3-2 任务状态图

Fig.3-2 State of Tasks of μC/OS-Ⅱ

睡眠态(Dormant)指任务驻留在程序空间之中，还没有交给μC/OS-Ⅱ管理。把任务交给μC/OS-Ⅱ管理是通过调用下述两个函数之一：OSTaskCreate()或OSTaskCreateExt()。当任务一旦建立，这个任务就进入就绪态准备运行。任务的建立可以是在多任务运行开始之前，也可以是动态地被一个运行着的任务建立。如果一个任务是被另一个任务建立的，而这个任务的优先级高于建立它的那个任务，则这个刚刚建立的任务将立即得到CPU的控制权。一个任务可以通过调用OSTsakDel()返回到睡眠态，或通过调用该函数让另一个任务进入睡眠态。

调用OSStart()可以启动多任务。OSStart()函数运行进入就绪态的优先级最高的任务。就绪的任务只有当所有优先级高于这个任务的任务转为等待状态，或者是被删除了，才能进入运行态。

正在运行的任务可以通过调用两个函数之一将自身延迟一段时间，这两个函数是OSTimeDly()或OSTimeDlyHMSM()。这个任务于是进入等待状态，等待这段

时间过去，下一个优先级最高的、并进入了就绪态的任务立刻被赋予了CUP的控制权。等待的时间过去以后，系统服务函数OSTimeTick()使延迟了的任务进入就绪态。

正在运行的任务期待某一事件的发生时也要等待，方法是调用以下3个函数之一：OSSemPend()，OSMboxPend()或OSQPend()。调用后，任务进入等待状态(Waiting)。当任务因等待事件被挂起(Pend)，下一个优先级最高的任务立即得到了CUP的控制权。当事件发生了，被挂起的任务进入就绪态。事件发生的报告可能来自另一个任务，也可能来自中断服务子程序。

正在运行的任务是可以被中断的，除非该任务将中断关闭，或者μC/OS-Ⅱ将中断关了。被中断了的任务就进入了中断服务态(ISR)。响应中断时，正在执行的任务被挂起，中断服务子程序控制了CPU的使用权。中断服务子程序可能会报告一个或多个事件的发生，而使一个或多个任务进入就绪态。在这种情况下，从中断服务子程序返回之前，μC/OS-Ⅱ要判定被中断的任务是否还是就绪态任务中优先级最高的。如果中断服务子程序使一个优先级更高的任务进入了就绪态，则新进入就绪态的这个优先级更高的任务将得以运行，否则原来被中断了的任务才能继续运行。当所有的任务都在等待事件发生或等待延迟时间结束，μC/OS-Ⅱ执行空闲任务(idle task)，执行S0TaskIdle()函数。

3.4.3 任务控制块

任务控制块是一个数据结构，当任务的CPU使用权被剥夺时，μC/OS-Ⅱ用它来保存该任务的状态。当任务重新得到CUP使用权时，任务控制块能确保任务从当时被中断的那一点丝毫不差地继续执行。OS_TCBS全部驻留在RAM中。

所有的任务控制块都放在一个任务控制块列表数组OSTCBTbl[]中，在μC/OS-Ⅱ初始化时，所有任务控制块OS_TCBs被链接成空任务控制块的单向链表。任务建立时，空任务控制块指针OSTCBFreeList指向的任务控制块便赋给了该任务。然后OSTCBFreeList的值调整为指向下链表中下一个空的任务控制块，一旦任务被删除，它的任务控制块就会回到空任务控制块链表中，如图3-3所示。

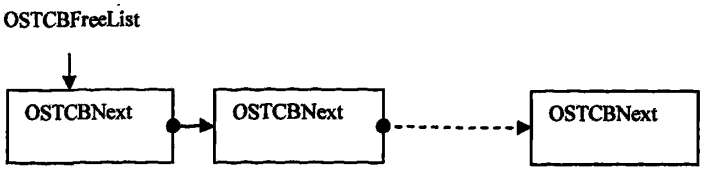

图3-3 空任务列表

Fig. 3-3 Free Task List

任务控制块代码如下：

```
typedef struct os_tcb {
    OS_STK          *OSTCBStkPtr;
#if OS_TASK_CREATE_EXT_EN
    void            *OSTCBExtPtr;
    OS_STK          *OSTCBStkBottom;
    INT32U          OSTCBStkSize;
    INT16U          OSTCBOpt;
    INT16U          OSTCBId;
#endif
    struct os_tcb *OSTCBNext;
    struct os_tcb *OSTCBPrev;
#if (OS_Q_EN && (OS_MAX_QS >= 2)) || OS_MBOX_EN || OS_SEM_EN
    OS_EVENT        *OSTCBEventPtr;
#endif
#if (OS_Q_EN && (OS_MAX_QS >= 2)) || OS_MBOX_EN
    void            *OSTCBMsg;
#endif
    INT16U          OSTCBDly;
    INT8U           OSTCBStat;
    INT8U           OSTCBPrio;
    INT8U           OSTCBX;
    INT8U           OSTCBY;
    INT8U           OSTCBBitX;
    INT8U           OSTCBBitY;
#if OS_TASK_DEL_EN
    BOOLEAN         OSTCBDelReq;
#endif
} OS_TCB;
```

OSTCBStrPtr是指向当前任务栈顶的指针。μC/OS-II允许每个任务有自己的

栈，每个任务栈的容量可以是任意的。有些商业内核要求所有任务栈的容量都一样，除非用户写一个复杂的接口函数来改变。这种限制浪费了RAM资源，当各个任务需要的栈空间不同时，也必须按任务中预期栈容量的最大值来分配栈空间。OSTCBStrPtr是OS_TCB数据结构中唯一的一个能用汇编语言来处置的变量(在任务切换段的代码Context-Switching Code之中)，把OSTCBStrPtr放在数据结构的最前面，使得从汇编语言中处理这个变量时较为容易。

OSTCBNext和OSTCBPrev用于任务控制块OS_TCBS的双重链接，该链表在时钟节拍函数OSTimeTick()中使用，用于刷新各个任务的任务延迟变量OSTCBDly，每个任务的任务控制块OS_TCB在任务建立的时候被链接到链表中，在任务删除的时候从链表中被删除。双重连接的链表使得任一成员都能被快速插入或删除。

OSTCBDly当需要把任务延时若干时钟节拍时要用到这个变量，或者需要把任务挂起一段时间以等待某事件的发生，这种等待是有超时限制的。在这种情况下，这个变量保存的是任务允许等待事件发生的最多时钟节拍数。如果这个变量为0，表示任务不延时，或者表示等待事件发生的时间没有限制。

OSTCBStat是任务的状态字。当其为0，任务进入就绪态。也可以赋其它的值表示任务的其他状态。

OSTCBPrio是任务优先级。这个值越小，任务的优先级越高。

OSTCBX、OSTCBY、OSTCBBitX和OSTCBBitY用于加速任务进入就绪态的过程或进入等待事件发生状态的过程。这些值是在任务建立时算好的，或者是在改变任务优先级时算出的，避免了在运行中去计算这些值。

OSTCBStkBottom指向任务栈底的指针。如果微处理器的栈指针是递减的，即栈存储器从高地址向低地址方向分配，则OSTCBStkBottom指向任务使用的栈空间的最低地址。类似地，如果微处理器的栈是从低地址向高地址递增型的，则OSTCBStkBottom指向任务可以使用的栈空间的最高地址。

OSTCBEventPtr是指向事件控制块的指针。

OSTCBMsg是指向传给任务的消息的指针。

3.4.4 任务创建

μC/OS-II管理用户的任务，用户必须要先建立任务。用户可以通过传递任务地址和其它参数到以下两个函数之一来建立任务：OSTaskCreate()或OSTaskCreateExt()。OSTaskCreateExt()与μC/OS是向下兼容的，OSTaskCreateExt()是OSTaskCreate()的扩展版本，提供了一些附加的功能。用两个函数中的任何一个都可以建立任务。任务可以在多任务调度开始前建立，也可以在其它任务的执行过程中被建立。在开始多任务调度(即调用OSStart()前，用户必须建立至少一个任务。任务不能由中断服务程序(ISR)来建立。

OSTaskCreate()需要四个参数：task是任务代码的指针，Pdata是当任务开始执行时传递给任务的参数的指针，ptos是分配给任务的堆栈的栈顶指针， Prio是分配给任务的优先级。任务建立函数伪码如下。

```
INT8U OSTaskCreate (void (*task)(void *pd), void *pdata, OS_STK *ptos, INT8U prio)
{
    void    *psp;
    INT8U   err;
    if (prio > OS_LOWEST_PRIO) {
(
        return (OS_PRIO_INVALID);
    }
    OS_ENTER_CRITICAL();
    if (OSTCBPrioTbl[prio] == (OS_TCB *)0) {
        OSTCBPrioTbl[prio] = (OS_TCB *)1;
        OS_EXIT_CRITICAL();
        psp = (void *)OSTaskStkInit(task, pdata, ptos, 0);
        err = OSTCBInit(prio, psp, (void *)0, 0, 0, (void *)0, 0);
        if (err == OS_NO_ERR) {
            OS_ENTER_CRITICAL();
            OSTaskCtr++;
            OSTaskCreateHook(OSTCBPrioTbl[prio]);
            OS_EXIT_CRITICAL();
            if (OSRunning) {
                OSSched();
            }
        } else {
```

```
            OS_ENTER_CRITICAL();
            OSTCBPrioTbl[prio] = (OS_TCB *)0;
            OS_EXIT_CRITICAL();
        }
        return (err);
    } else {
        OS_EXIT_CRITICAL();
        return (OS_PRIO_EXIST);
    }
}
```

OSTaskCreate()一开始先检测分配给任务的优先级是否有效。任务的优先级必须在0到OS_Lowest_Prio之间。接着，OSTaskCreate()要确保在规定的优先级上还没有建立任务。在使用μC/OS-Ⅱ时，每个任务都有特定的优先级。如果某个优先级是空闲的，μC/OS-Ⅱ通过放置一个非空指针在OSTCBPrioTbl[]中来保留该优先级，之后初始化系统中任务的结构。

3.4.5 任务就绪表

μC/OS-Ⅱ管理了一个任务就绪表，就绪表就是一个矩阵，OSRdyTbl矩阵位中的值为0或者1，表示对应的prio任务是否就绪。同样，纵坐标的一组位（也就是一个字节）合称一个任务组。可以看到prio的数据分为两部分，一部分Y表示的是纵坐标，X表示的是横坐标。OSRdyGrp是纵坐标上就绪任务组的记录，只要该组中任何位代表的任务就绪，OSRdyGrp纵坐标的对应位就标记为就绪。如果要知道该组中具体哪个横坐标位就绪，需要再查一次横坐标的具体情况。

OSRdyGrp和OSRdyTbl之间的关系如图3-4，且满足规则：若OSRdyTbl[i]中任何一位被置1，则OSRdyGrp的第i位置1。

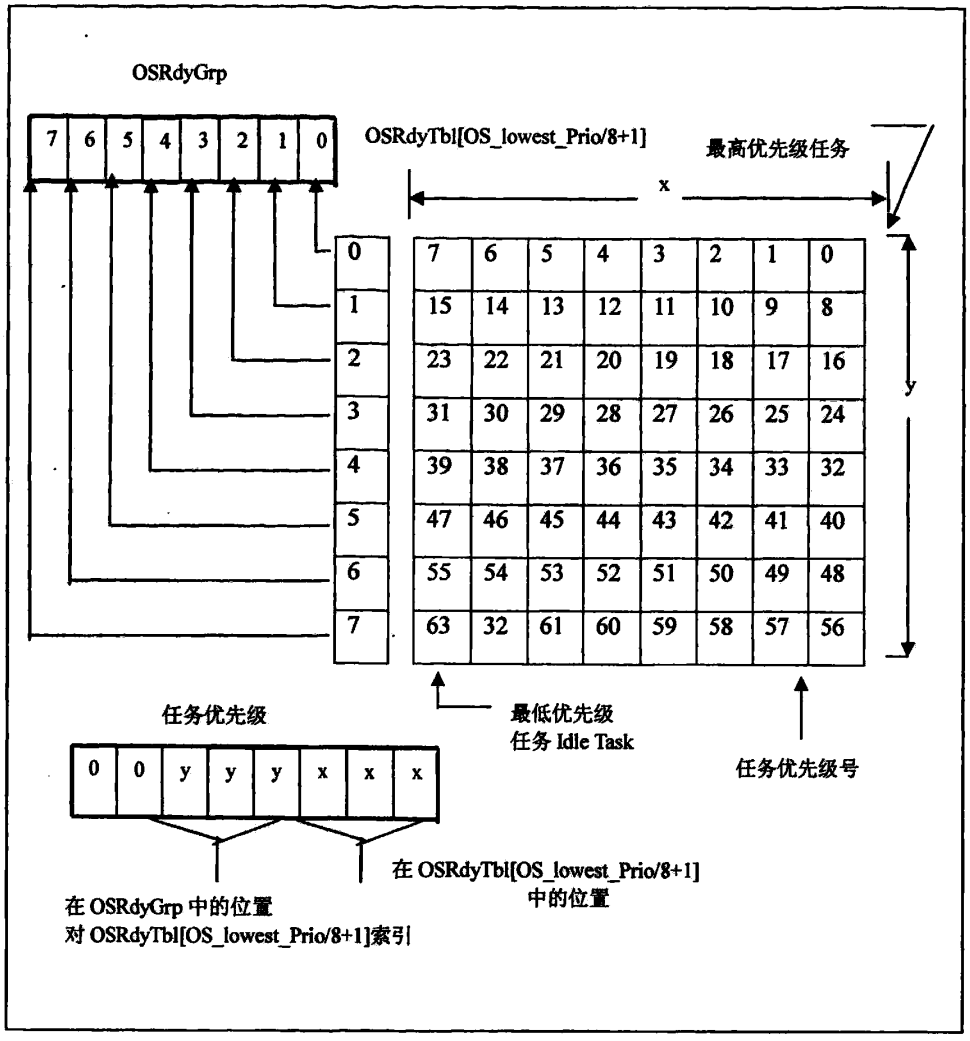

图3-4 μC/OS-Ⅱ就绪表

Fig.3-4 Ready List of μC/OS-Ⅱ

μC/OS-Ⅱ的64个任务按优先级被分为8组，每组8个任务。每个任务被赋予不同的优先级等级，从0级到最低优先级OS_LOWEST_PRIO，包括0和OS_LOWEST_PRIO在内。当μC/OS-Ⅱ初始化的时候，最低优先级OS_LOWEST_PRIO总是被赋给空闲任务Idle Task。就绪表OSRdyTbl[]数组的大小取决于OS_LOWEST_PRIO。

对于μC/OS-Ⅱ就绪表，可以用下列代码将任务放入就绪表。(假定prio是被操作任务的优先级)：

```
OSRdyGrp              |=OSMapTbl[prio»3]; //用Y映射出纵坐标的位置
OSRdyTbl[prio»3]   |=OSMapTbl[prio&Ox07];//用X映射出横坐标的位置
```

3.4.6 任务调度

任务调度主要是协调任务对计算机系统资源(如内存、I/O设备、CPU)的争夺使用。进程调度又称为CPU调度，其根本任务是按照某种原则为处于就绪状态的进程分配CPU。进程调度可分为“抢占式调度”和“非抢占式调度”两种基本方式。所谓“非抢占式调度”是指：一旦某个进程被调度执行，则该进程一直执行下去直至该进程结束，或由于某种原因自行放弃CFU，进入等待状态才将CPU重新分配给其它进程。所谓“抢占式调度”是指：一旦就绪状态出现优先级更高的进程，或者当前运行的进程已用满规定的时间片时，便立即剥夺当前进程的运行使其设置为就绪状态，同时把CUP时间分配给其它更高优先级的就绪进程。

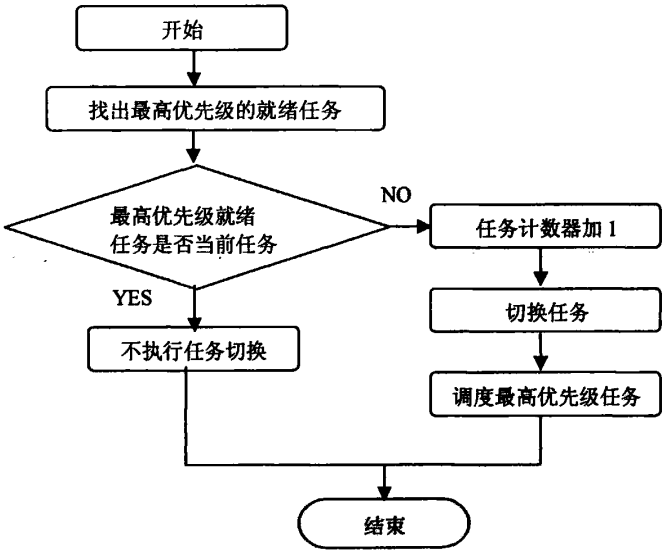

图3-5 任务调度流程图

Fig.3-5 Task Schedule Flow Chart

μC/OS- II 总是运行进入就绪态任务中优先级最高的那一个。确定哪个任务优先级最高以及确定下一个应该运行的任务是由调度器(Scheduler)完成的。任务级的调度是由函数OSSched()完成的。

如果任务调度是允许的，即没有上锁，则任务调度函数将找出那个进入就绪态且优先级最高的任务，进入就绪态的任务在就绪任务表中有相应的位置。一旦找到那个优先级最高的任务，OSSched()检验这个优先级最高的任务是不是当前正在运行的任务，以此来避免不必要的任务调度。为实现任务切换，

OSTCBHighRdy必须指向优先级最高的那个任务控制块OS_TCB，这是通过以OSPrioHighRdy为下标的OSTCBPrioTbl[]数组中的那个元素赋给OSPrioHighRdy来实现的。接着，统计计数器OSCtxSwCtr加1，以跟踪任务切换次数。最后宏调用OS_TASK_SW()来完成实际上的任务切换。任务调度的流程如图3-5所示。

3.4.7 任务切换

任务开始后需要按照设定的要求进行任务切换。等待特定条件、获得条件、将自身挂起、中断都可引起任务切换。这里的条件是指信号量、互斥信号量、事件标志、消息邮箱、消息队列等。任务切换分为任务级任务切换和中断级任务切换。任务级切换由OSSched()完成，它一般由实现通信功能的等待资源、释放资源函数如OSSemPend()、OSSemPost()等、延时函数OSTimeDly()等以及给任务调度开锁函数OSSchedUnlock()调用。它找出优先级最高的任务，然后调用OS_TASK_SW()完成切换。

中断级的切换由OSIntExit()函数实现，它一般由中断服务程序调用，包括时钟节拍中断服务程序。它调用OSIntCtxSw()实现任务切换。OSIntCtxSw的工作原理和OS_TASK_SW类似，只不过在中断服务程序中已经保存了寄存器，就不用再像OS_TASK_SW那样首先保存寄存器。

3.5 本章小结

本章从μC/OS-Ⅱ相对于其他操作系统的优点和特点入手，主要对其内核结构进行了分析研究。重点介绍了任务管理模块，对用户任务结构、任务状态、任务控制块和任务就绪表、任务创建和任务调度和切换进行了深入分析。

第四章 μC/OS-Ⅱ实时操作系统的可移植性研究

4.1 概述

所谓移植，就是使一个实时内核能在某个微处理器或微控制器上运行。要想使用μC/OS-Ⅱ，应该把它移植到处理器芯片上。μC/OS-Ⅱ的特点决定了它具有很好的可移植性，但它也包含了一些与处理器相关的代码，这是因为μC/OS-Ⅱ在读、写处理器的一些寄存器时只能通过汇编语言来实现。μC/OS-Ⅱ从结构组成上可以分成核心部分模块、任务管理模块、任务间同步、时间管理模块、与通信模块、内存管理模块、移植相关模块这五个部分。其中核心代码与处理器无关，而和应用相关的配置代码及与处理器相关的移植代码却要根据需要重写。

μC/OS-Ⅱ运行条件如下：

- 处理器支持的C编译器能产生可重入代码；
- 处理器支持中断，并且能产生定时中断，通常为10～100Hz；
- 用C语言就可以开、关中断；
- 处理器支持能容纳一定数量的硬件堆栈；
- 处理器有将堆栈指针和其他CPU寄存器的内容读出并存储到堆栈或内存中的指令

4.2 S3C44B0 简介

本文采用北京恒丰科技有限公司的开发板HF44B0，如图4-1所示。HF44B0开发板是基于三星公司S3C44B0x高性能ARM处理器的嵌入开发平台，其硬件规格如下：

- Samsung S3C44B0X 66MHz(ARM7内核)；
- 8M SDRAM；
- 2M FLASH；
- 10M以太网接口；
- IDE接口；
- USB1.1接口；

- 音频输出；
- MIC输入接口；
- 时实时钟，备有可充电电池；
- IIC总线接口；
- 实时系统时钟；
- LCD接口：支持640*480以下单色或320*240以下STN/DSTN 256色；
- 2个RS232串行口；
- 20针JTAG调试端口；
- 1*4键盘；
- 可扩展的总线接口,CPU的160个引脚全部用2.54标准插座引出。

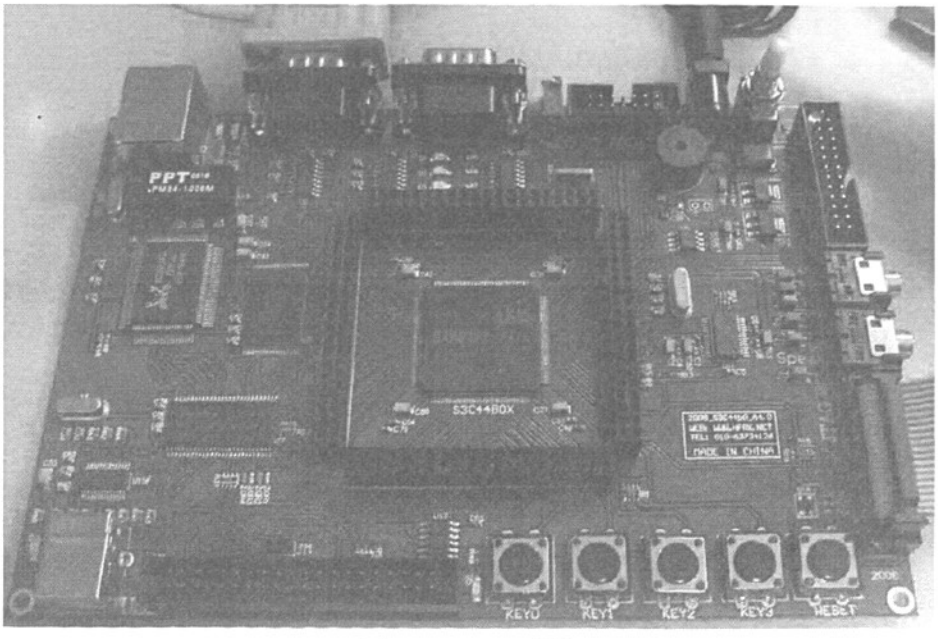

图4-1 HF44B0开发板

Fig. 4-1 HF44B0 Development Board

Samsung公司推出的16/32位RISC处理器S3C44B0X为手持设备和一般类型应用提供了高性价比和高性能的微控制器解决方案。为了降低成本S3C44B0X提供了丰富内置部件：8KB Cache，可选的内部SRAM，LCD控制器，带自动握手的2通道UART，4通道DMA，系统管理器（片选逻辑，FP/EDO/SDRAM控制器），带PWM功能5通道定时器，I/O端口，RTC，8通道10位ADC，IIC总线接口，IIS总线接口，同步SIO接口和PLL倍频器。S3C44B0X采用ARM7TDMI内核，0.25μm

工艺的CMOS标准宏单元和存储编译器。它低功耗、精简、出色和全静态的设计特别适用于对成本和功耗敏感的应用。同样S3C44B0X还采用了一种新的总线结构，即SAMBAII（SAMSUNGARM CPU嵌入式微处理器总线结构）。

S3C44B0X的CPU核是由ARM公司设计的16/32位ARM7TDMI RISC处理器（66MHZ）。ARM7TDMI体系结构的特点是它集成了Thumb代码压缩器，一个片上ICE断点调试支持，和一个32位的硬件乘法器。

S3C44B0X通过提供全面的、通用的片上外设，大大减少了系统电路中除处理器以外的元器件配置，从而最小化系统的成本。片上集成的主要功能如下：

- 2.5VARM7TDMI内核，带有8K Cache(主频高至66MHz)；
- 外部存储器控制器（FP/EDO/SDRAM控制，片选逻辑）；
- LCD控制器(最大支持256色DSTN)，并带有1通道LCD专用DMA；
- 2通道通用DMA、2通道外设DMA并具有外部请求引脚；
- 2通道UART带有握手协议(支持IrDA1.0，具有16-byte FIFO)/1通道SIO；
- 1通道多主IIC-BUS控制器；
- 1通道IIS-BUS控制器；
- 5个PWM定时器和1个内部定时器；
- 看门狗定时器；
- 71个通用I/O口/8通道外部中断源；
- 功耗控制：具有正常、低速、空闲和停止模式；
- 8通道10位ADC；
- 具有日历功能的RTC；
- 带PLL的片上时钟发生器。

4.3 μC/OS-Ⅱ在S3C44B0上的移植

第三章的图3-1介绍了μC/OS-Ⅱ文件组织与硬件的关系。如果理解了处理器和C编译器的技术细节，移植μC/OS-Ⅱ的工作实际上是并不复杂。前提是处理器和编译器满足了μC/OS-Ⅱ的要求，并且已经有了必要工具。

在本论文中所作的移植工作，包括以下几个内容：

- OS_CPU.H中需要针对具体处理器的字长重新定义一系列数据类型；
- OS_CPU.H中需要设置常量来标志堆栈增长方向；

- OS_CPU.H中需要声明用于开关中断和任务切换的宏；
- OS_CPU_A.ASM中需要改写4个汇编语言的函数；
- OS_CPU_C.C中需要用C语言写6个简单的函数。

4.3.1 改写 OS_CPU.H

(1)定义数据类型

```
typedef unsigned char     INT8U;     //8位无符号整数
typedef signed char       INT8S;     //8位有符号整数
typedef unsigned short    INT16U;    //16位无符号整数
typedef signed short      INT16S;    //16位有符号整数
typedef unsigned long     INT32U;    //32位无符号整数
typedef signed long       INT32S;    //32位有符号整数
typedef float             FP32;      //单精度浮点数
typedef double            FP64;      //双精度浮点数
```

(2)定义使能和禁止中断宏

```
#define OS_ENTER_CRITICAL() ARMDisableInt()
#define OS_EXIT_CRITICAL()  ARMEnableInt()
ARMDisableInt:
STMDB    sp!, {r0,lr}        /*保存r0，lr*/
MRS   r0, CPSR            /*取状态寄存器内容到r0*/
ORR   r0, r0, #0xc0       /*FIQ、IRQ相应位置1，关闭中断*/
MSR       CPSR_cxsf, r0
LDMIA   sp!, {r0,pc}      /*恢复相关寄存器*/
ARMEnableInt:
STMDB   sp!, {r0,lr}      /*保存r0，lr*/
MRS   r0, CPSR            /*取状态寄存器内容到r0*
BIC   r0, r0, #0xc0       /*FIQ、IRQ相应位置0，使能中断*/
MSR   CPSR_cxsf, r0
LDMIA   sp!, {r0,pc}      /*恢复相关寄存器*/
```

(3)定义中断增长方式

#define OS_STK_GROWTH 1; /*堆栈从高地址向低地址增长*/

(4)OS_TASK_SW()

这里直接用汇编语言编写OS_TASK_SW()，而不是OSCtxSw()，并使它完成任务栈的保存和加载的功能。在OS_CPU_A.ASM中给出其汇编代码。

(5)浮点函数 S3C44B0X没有硬件浮点运算单元，不需要编写浮点函数。

4.3.2 改写 OS_CPU_C.C 文件

这里我们只需要任务堆栈初始化函数OSTaskStkInit()，其它的函数写成空函数就可以了。

```
void *OSTaskStkInit (
void (*task)(void *pd), void *pdata, void *ptos, INT16U opt){
unsigned int *stk ;
opt = opt;                    /*暂时没有使用，这样写防止编译时出错*/
stk=(unsigned int *)ptos;     /*加载栈指针*/
/*为新任务创建一个上下文环境*/
*--stk = (unsigned int) task;  /*pc*/
*--stk = (unsigned int) task;  /*lr*/
*--stk = 0;                    /*r12*/
*--stk = 0;                    /*r11*/
*--stk = 0;                    /*r10*/
*--stk = 0;                    /*r9*/
*--stk = 0;                    /*r8*/
*--stk = 0;                    /*r7*/
*--stk = 0;                    /*r6*/
*--stk = 0;                    /*r5*/
*--stk = 0;                    /*r4*/
*--stk = 0;                    /*r3*/
*--stk = 0;                    /*r2*/
*--stk = 0;                    /*r1*/
*--stk = (unsigned int) pdata;      /*r0*/
*--stk = (SVC32MODE|0x0);           /*cpsr  IRQ, FIQ 禁止*/
*--stk = (SVC32MODE|0x0);           /*spsr  IRQ, FIQ 禁止*/
```

```
return ((void *)stk);}
```

4.3.3 改写 OS_CPU_A.ASM 文件

(1)编写OSStartHighRdy

```
OSStartHighRdy:
BL OSTaskSwHook                 /* 调用用户Hook函数*/
MOV   R0,#1
LDR   R1,=OSRunning
STRB   R0,[R1]                   /* OSRunning=True;多任务开始*/
LDR    r4, addr_OSTCBCur        /* 获得当前任务控制块地址*/
LDR    r5, addr_OSTCBHighRdy    /* 获得最高优先级任务控制块地址*/
LDR    r5, [r5]                 /* 获得堆栈指针*/
LDR    sp, [r5]                 /* 切换到新堆栈*/
STR    r5, [r4]                 /* 设置新的当前任务控制块地址*/
LDMFD sp!, {r4}
MSR    SPSR_cxsf, r4
LDMFD sp!, {r4}                 /* 从栈顶获得新的状态*/
MSR    CPSR_cxsf, r4
LDMFD sp!, {r0-r12, lr, pc }    /* 开始新任务*/
```

(2)编写OS_TASK_SW

```
OS_TASK_SW:
STMFD   sp!, {lr}        /* 保存pc */
STMFD   sp!, {lr}        /* 保存lr */
STMFD   sp!, {r0-r12}        /* 保存寄存器和返回地址 */
MRS     r4, CPSR
STMFD   sp!, {r4}        /* 保存状态寄存器CPSR */
MRS     r4, SPSR
STMFD   sp!, {r4}        /* 保存状态寄存器SPSR */
/* 设当前任务为最高优先级任务 */
LDR     r4, addr_OSPrioCur
LDR     r5, addr_OSPrioHighRdy
```

```
LDRB    r6, [r5]
STRB    r6, [r4]
/* 获得当前任务控制块TCB地址 */
LDR     r4, addr_OSTCBCur
LDR     r5, [r4]
STR     sp, [r5]          /* 保存堆栈指针到其TCB*/
/* 获得高优先级任务TCB地址 */
LDR     r6, addr_OSTCBHighRdy
LDR     r6, [r6]
LDR     sp, [r6]          /* 获得新任务的指针*/
/* 最高优先级任务TCB为当前任务TCB*/
STR     r6, [r4]          /* 设置新的当前任务TCB地址*/
LDMFD sp!, {r4}           /* 寄存器顺序出栈*/
MSR     SPSR_cxsf, r4
LDMFD sp!, {r4}
MSR     CPSR_cxsf, r4
LDMFD sp!, {r0-r12, lr, pc}
```

(3)编写OSIntCtxSw

OSIntCtxSw的工作原理和OS_TASK_SW类似，只不过在中断服务程序中已经保存了寄存器，就不用再像OS_TASK_SW那样首先保存寄存器。

(4)编写OSTickISR

```
OSTickISR:
STMDB     sp!,{r0-r11,lr}         /* 保存寄存器*/
/* 中断禁止（不是必要的）*/
MRS     r0, CPSR
ORR     r0, r0, #0x80     /* IRQ置1，禁止状态*/
MSR     CPSR_cxsf, r0
LDR       r0,=0x4000000
BL        SysDISInterrupt
/* 中断结束，清除中断挂起位*/
LDR     r0, =I_ISPC
LDR     r1, =BIT_TIMER0
```

```
STR    r1, [r0]
BL   IrqStart
BL   OSTimeTick
BL   IrqFinish          /* 返回值为0，不进行任务切换，为1，切换*/
CMP    a1, #0
LDRNE   pc, =_CON_SWAP    /* 进行任务切换*/
```

4.4 μC/OS-II的初始化

在使用μC/OS-II之前，需要对其进行初始化，μC/OS-II调用OSInit()进行初始化。它初始化μC/OS-II所有的变量和数据结构，并建立空闲任OS_TaskIdle()。空闲任务的优先级设为最低（也就是OS_LOWEST_PRIO），并且处于就绪状态。如果将相关参数设置为允许，μC/OS-II还将创建统计任务OS_TaskStat()。

μC/OS-II在初始化时主要创建了一个任务控制块的双向链表。刚初始化时，这个链表只有空闲任务和统计任务两个任务。以后每当创建一个任务就将任务的TCB加入这个链表的表头(表头指针是OSTCBList)。除此以外，μC/OS-II还初始化了五个空的数据结构缓冲区。它们分别是任务控制块缓冲区、事件控制块缓冲区、消息队列缓冲区、标志控制块缓冲区和存储控制块缓冲。每个缓冲区中包含了相应的一定数目的控制块数据结构，每当创建这些控制块时，需要首先向这些缓冲区申请一个控制块。

4.5 多任务启动

和其它C语言一样，μC/OS-II程序运行首先从main()函数开始，其main()函数的形式如下：

```
void main(void){
  OSInit();     /*初始化μC/OS-II */
  OSTaskCreate()或OSTaskCreateExt();/*创建任务*/
  OSStart();  /*多任务开始*/
}
```

具体工作过程如下：

- μC/OS-II在ucos_ii.h中声明任务控制块OS_TCB和任务优先级表*OSTCBPrioTbl[OS_LOWEST_PRIO+1]；在OSInit()中初始化任务控制块缓冲区

OSTCBTbl[OS_MAX_TASKS]。

● 调用OSTaskCreate()或OSTaskCreateExt()创建任务，这两个函数调用OSTCBInit()将创建的任务控制块放入OSTCBPrioTbl：

```
ptcb = OSTCBFreeList;          /* 得到一个空闲TCB控制块*/
OSTCBPrioTbl[prio] = ptcb;  /*将空闲TCB加入优先级表 */
```

● OSStart()开始整个任务调度

```
y = OSUnMapTbl[OSRdyGrp];      /* 找出优先级最高的优先级 */
x = OSUnMapTbl[OSRdyTbl[y]];
OSPrioHighRdy = (INT8U)((y << 3) + x);
OSPrioCur          = OSPrioHighRdy;
OSTCBHighRdy = OSTCBPrioTbl[OSPrioHighRdy];
/* 使优先级最高的任务准备运行 */
OSTCBCur          = OSTCBHighRdy;
OSStartHighRdy();
```

● 最后调用汇编函数OSStartHighRdy，通过将最高优先级任务的堆栈内容赋予相应的寄存器完成任务的开始。

```
OSStartHighRdy:
调用OSTaskSwHook();
OSRunning=TRUE;
得到要运行的任务的指针;
从任务堆栈中恢复所有寄存器;
返回;
```

4.6 μC/OS-Ⅱ移植测试

4.6.1 调试和下载环境

针对ARM处理器核的C语言编译器有很多，如SDT，ADS，IA，RTASKING和GCC等。据了解，目前在国内最流行的是SDT，ADS、和GCC。SDT和ADS均为ARM公司自己开发的，ADS为SDT的升级版，以后ARM公司不再支持SDT，所以不需要选择SDT。GCC虽然支持广泛，很多开发套件使用它作为编译器，但其编译效率与ADS比起来实在太低，初步比较同一个C程序速度相差2-3倍，这对

充分发挥芯片性能极其不利，所以最终选择ADS编译和调试程序。

ADS1.2(ARM Developer Suite v1.2)是一个使用方便的集成开发环境，是由ARM公司提供的专门用于ARM相关应用开发和调试的综合性软件。在功能和易用性上比SDT都有提高，是一款功能强大又易于使用的开发工具。

ADS主要由以下部件构成：命令行开发工具、图形界面开发工具、各种辅助工具、支持软件。

1)命令行开发工具

Armcc：ARM的C编译器，支持标准C语言；

ArmePP：ARMC++编译器，将ISOC++或EC++编译成32位ARM指令代码；

Tcc：Thumb的C编译器，支持标准C语言；

TcpP：Thumb的C编译器，支持ISOC++以及EC++语言；

Armasm：ARM和Thumb的汇编器，支持ARM以及Thumb汇编语言；

Armhnk：ARM的链接器，产生ELF格式文件；

Armds：ARM以及Thumb的调试器，支持代码级调试，可通过单步执行、设置断点以及检查变量或内存。

2)ARM的图形界面开发工具包括：AXD(Windows以及UNIX下的ARM调试器)、ADW(Windows下的ARM调试器)、ADU(UNIX下的ARM调试器)、CodeWarriorIDE(Windows下的项目管理工具)。

3)实用工具

FromELF：ARM映像文件转换工具。该命令将ELF格式的文件作为输入文件，将该格式转换为各种输出格式的文件；

Armar：ARM库函数生成器将一系列ELF格式的目标文件以库函数的形式集合在一起，用户可以把一个库传递给一个链接器以代替几个ELF文件；

Flashdownloader：将二进制映像文件下载到ARM开发板的Flash存储器；

ArmProf：显示程序调试时的信息文件。

4)支持的软件

ARMulator：ARM指令集仿真器，集成在ARM的调试器AXD中，它提供对ARM处理器的指令集的仿真，为ARM和Thumb提供精确的模拟；

Angel：运行在目标机上的调试监控程序，可以对ARM、Thumb代码的应用程序调试。

4.6.2 μC/OS-Ⅱ移植测试

编写好移植代码后，使用ADS1.2下载移植代码。ADS1.2编译、链接，生成可调试的AXF文件。通过JTAG下载此AXF文件至目标板。代码移植结束后，下一步工作就是测试。测试可以在没有应用程序的情况下进行。也就是让内核自己测试自己。这样做有两个好处：第一，避免使本来就复杂的事情更加复杂；第二，如果出现问题，可以知道问题出在内核代码上而不是应用程序。刚开始的时候可以运行一些简单的任务和时钟节拍中断服务例程。一旦多任务调度成功地运行了，再添加应用程序的任务。

```
#include "includes.h"
 int main(viod)
 {
   OSInit();
   OSStart();
 }
```

这是没有添加任何的应用任务，只是让系统内核运行起来，仅仅检查移植代码的各种语法错误。

4.7 本章小结

本章首先提出了移植的概念和μC/OS-Ⅱ的运行条件。进而介绍了所用的S3C44B0开发板，主要研究了μC/OS-Ⅱ在该开发板的移植。对内核的改造包括：OS_CPU.H中需要针对具体处理器的字长重新定义一系列数据类型、OS_CPU.H中需要设置常量来标志堆栈增长方向、OS_CPU.H中需要声明用于开关中断和任务切换的宏、OS_CPU_A.ASM中需要改写4个汇编语言的函数、OS_CPU_C.C中需要用C语言写6个简单的函数。研究了μC/OS-Ⅱ初始化和多任务的启动。最后介绍了调试和下载环境，并对移植进行了测试。

第五章 基于μC/OS-Ⅱ的实时多任务调度算法研究

5.1 调度算法的改进

调度算法即用以确定所要调度的任务能够在某一特定时刻执行的规则集。

一个良好的任务调度算法应该主要体现在以下几个方面：

(1)公平：保证每个进程得到合理的CPU时间；

(2)高效：使CPU保持忙碌状态，即在CPU上总有进程在运行；

(3)响应时间：使用户交互的响应时间尽可能短；

(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进程尽可能多。

5.1.1 单调速率调度

任务按照单调速率优先级分配（RM：Rate Monotonic Priority Assignment）的调度算法称为单调速率调度。单调速率调度优先级分配是指任务的优先级按照任务的周期T来分配。周期短的任务优先级高；反之，周期长的任务优先级低，调度程序总是试图最先运行周期最短的任务。单调速率调度算法由Liu和Layland于1973年引入的，是首先为实时系统开发的用于抢占式的硬实时周期任务调度的静态优先调度算法，已被证明是静态优先级调度算法中最优的，成为其他静态调度算法事实上的应用研究基础，现在仍然被广泛使用。为了提高RM算法的处理机利用率，人们提出了若干改进方法，典型的有：子任务法、双优先级法和最优双优先级法。

RM调度算法主要内容：每个任务根据它的周期被指定一个固定的优先级，周期越短，优先级越高(即对两个任务i和j，$Ti<Tj<\rightarrow Pi>Pj$)。

单调速率调度的理论基础是单调速率分析(RMA：Rate Monotonic Analysis)，RMA是基于一系列假设的单调速率调度模型的，基本假设如下：

- 所有的任务请求都是周期性的，并具有硬期限要求；
- 任务的期限要求仅限于任务必须在任务的下一个请求发生之前完成，即

$D_i=T_i$，$i=1$，…，n；

- 任务都是相互独立的，每个任务的请求不依赖于其他任务的请求；
- 每个任务的运行时间是不变的，运行时间是处理器在无中断情况下用于处理该任务的时间；
- 调度程序本身和任务切换时间忽略不计；
- 任务之间是可抢占式的；
- 所有任务的分配都在单处理器上进行。

RM算法可调度性判断充分条件：

定理5-1：设一任务集$S=\{\tau_1,\tau_2,\ldots,\tau_n\}$，相应的任务周期分别是$T_1, T_2,\ldots, T_n$，执行时间为$c_1, c_2, \ldots, c_n$，认为集$S$的处理器利用率为$U$，$S$可以被RMS调度，如果[39]

$$U=\sum_{i=1}^{n}(c_i/T_i)\le n(2^{1/n}-1) \tag{5.1}$$

根据公式(5.1)可以得到处理机利用率界限表，如表5-1所示。

表5-1 处理机利用率界限表

Tab. 5-1 Table of processor utilization ratio limit

n	利用率界限
1	1.000
3	0.779
5	0.743
7	0.728
9	0.720
10	0.718
100	0.695
∞	0.693

表5-1给出了不同任务数情况下处理机的利用率值。n较小时利用率的界限为1；当n很大时，界限接近0.693。对于一个含有n个任务的任务集的实时系统，如果该系统是可调度的，那么各个实时任务必须在执行的每个周期都满足它的最终时限(即绝对截止期)。实时任务的某次执行必须在本次的最终期限到来之前完成，否则就不满足可调度性的条件。

例1：设有五个任务$\tau_1,\tau_2,\tau_3,\tau_4,\tau_5$，参数分别为：

τ_1：$c_1=10$，$T_1=100$

τ_2：$c_2=20$，$T_2=150$

τ_3: c_3=30，T_3=180

τ_4: c_4=40，T_4=240

τ_5: c_5=50，T_5=300

CPU的利用率U=10/100+20/150+30/180+40/240+50/300=0.733，由表5-1的数据可得五个周期性任务的处理器利用率界限为5（$2^{1/5}$-1）≈0.743>U，所以这五个任务可被RM算法调度。

Liu和Layland在文献[39]中证明了单调速率调度算法是最优的，即对于在任务其他静态优先级算法下的可调度的任务集，在RM算法下也是可调度的。

证明思想如下：假设一个任务集S采用其他静态优先级算法可以调度，设τ_i和τ_j是其中两个优先级（P）相邻的任务，$P_i \geqslant P_j$，假设$T_i > T_j$。将τ_i和τ_j的优先级互换（即$P_i \leqslant P_j$），可以证明这时S仍然可以调度。按照此方法，其他任何静态优先级调度最终都可以转换成单调调度速率调度。这样便证明了单调速率调度算法在所有静态优先级算法中是最优的。

定理5-1也称为利用率限定定理，是一个充分条件，或是悲观条件，对于一个任务集S，如果其n个任务的处理器利用率$U \leq n(2^{1/n}-1)$，则这n个任务一定是可以调度的；如果U≥1，则这n个任务一定是不可以调度的；如果$n(2^{1/n}-1) \leq U < 1$，则定理5-1并不能判定这n个任务一定不可以调度。基于定理5-1的单调速率调度算法通常被称为基本的单调速率调度算法。在此基础上，通过对单调速率调度模型的基本假设进行条件放宽或进一步严格化，产生了更多扩展型单调速率调度算法[40]。

RMS算法可调度的充分必要条件为：

定理5-2：设一任务集$S=\{\tau_1, \tau_2, \ldots, \tau_n\}$，相应的任务周期分别是$T_1, T_2, \ldots, T_n$，且$T_n > T_{n-1} > T_1$，执行时间为$c_1, c_2, \ldots, c_n$，S可以被RMS算法调度，当且仅当对每个任务$\tau_i$[41]：

$$\forall i, 1 \leq i \leq n, \min_{(k,l) \in Ri} \sum_{j=1}^{i} c_j \frac{1}{lT_k} \left\lceil \frac{lT_k}{T_j} \right\rceil = U = \sum_{i=1}^{n} (c_i / T_i) \leq 1 \qquad (5.2)$$

其中，$Ri = \{(k,l) \mid 1 \leq k \leq i, l = 1, \ldots, \lfloor T_i / T_k \rfloor\}$

5.1.2 期限最近者优先调度

期限最近者优先调度（EDF：Earliest Deadline First）调度算法，也称为死线驱动调度算法（DDS：Deadline Driven Scheduling）算法。EDF（DDS）以最后期限的顺序指定优先级，优先级最高的任务是距离最后期限最近的任务，优先级最低的任务是距离最后期限最远的任务，因此在每一个任务结束时，余下任务的优先级都必须重新计算。同RM算法较低的处理器利用率（上限69%）相比较，EDF可以达到100%的处理器利用率，并且如果EDF系统过载或者错过了期限，那么在错过最后期限之前，它将以100%的负荷运行。

EDF算法可调度的充分必要条件为：

定理5-3：设一任务集$S=\{\tau_1,\tau_2,\ldots,\tau_n\}$，相应的任务周期分别是$T_1, T_2,\ldots, T_n$，且$T_n>T_{n-1}>T_1$，执行时间为$c_1, c_2, \ldots, c_n$，$S$可以被EDF算法调度，当且仅当[42]

$$U=\sum_{i=1}^{n}(c_i/T_i)\le 1 \qquad (5.3)$$

EDF调度算法最大的优势在于，对于任何给定的任务集，只要处理器的利用率不超过100%，就能够保证它的可调度性[43]。EDF作为一种动态优先级调度算法是最优的，即对于在任何其他动态优先级算法下可调度的任务集，在EDF算法下也是可以调度的。

5.1.3 RM 和 EDF 比较

RM和EDF算法那个更好，取决于解决问题的标准。

RM：调度决策运行开销小，可预测性强，在瞬时过载的情况下可以保持稳定，即使系统在相对较短的时间内处于过载状态，那些具有最高优先级的任务（周期最短的任务）仍可以满足最后期限[44]，算法实现简单。这类调度算法适合于问题需求确定，并且运行中不会有较大变化情况。但是当处理器负载增加时，优先级较低的任务偶尔会超时，灵活性相对差些，不适合于动态变化或者不可预测环境下的任务调度。CPU利用率较低，但是它可以保证所有进程的最后期限都可以满足。权威的实时系统杂志1997年第3期（Real-Time Magazine, issue97-3）的一份调查报告表明，当时所占市场份额比较高的50个实时操作系统有41个都支持这种固定优先级抢占式调度算法[45]，本文所采用的μC/OS-II操作系统同样支持这种

算法。事实证明，这种固定优先级的做法在许多场合下调度多任务的效率已经足够了[46]。

EDF：在处理器利用率方面有较大的优势，可达到100%，调度算法比较灵活，能够应对变化的运行环境，适合于任务不断生成，并且在任务生成之前，其特性并不太清楚的动态实时系统。但是，相对于静态优先级算法，任务的优先级不确定，因此不易协调相互关联的多个任务，并且调度决策的运行开销相对比较大。在系统发生瞬间过载的情况下，也难以保证某些关键的硬实时任务的时间约束要求，很难诊断出即将出现过载的可能性，该算法的实现要比RMS的实现复杂的多。当系统的负载相对较低时，这种算法非常有效，但在系统负载较重时，系统性能会急剧下降，引起大量任务错过其最后期限，甚至可能导致CPU时间大量花费在调度决策上，因此系统的性能还不如FIFO方法[47]。

5.2 算法改进

5.2.1 算法思路

μC/OS-II的任务调度是按抢占式多任务系统设计的，规定所有任务的优先级必须不同，任务的优先级也唯一标识了该任务。即使是同等重要的任务也必须有优先级的差别，也就是说高优先级的任务在处理完成后必须进入等待或者挂起状态，否则低优先级的任务就不可能执行。在此选择了单调速率调度算法，并假设所有的任务都是周期性的。

将μC/OS-II中的任务划分为3个层次：实时任务、分时任务、后台任务。实时任务保留原本设计的绝对优先级调度，对系统驱动或者通信等实时性要求高的场合提供支持，任务创建接口保持不变。分时任务空间采用单调速率调度算法，各种任务依照周期大小依次执行（周期小的优先级高），适应于事物性处理和实时性要求不高的场合，并对此空间任务的创建和销毁提出了新的编程接口。后台任务是指IDLE、统计任务等在任务的空闲时间运行的任务，其在实时任务和分时任务的都没有就绪时才有机会运行，此区间也采用绝对优先级调度。

将OSRdyTbl[2]中的8个优先级为16-23的任务空间定义为分时任务空间，采用单调速率调度。当就绪队列中优先级没有比分时任务空间中的优先级高的时候，才进入单调速率调度。如果实际需要的分时任务空间比较大，可以进一步扩

充这个任务空间。

单调速率调度算法是静态调度算法，需要事先由用户按照任务的周期给每个任务分配固定的优先级。μC/OS-II也是静态调度，因此用户在设定好优先级后将系统中的OSTaskChangePrio()函数屏蔽就可以，无需对内核做较大的改变。

函数优先级分配的方法是按照任务的周期分配的，流程图如图5-1。

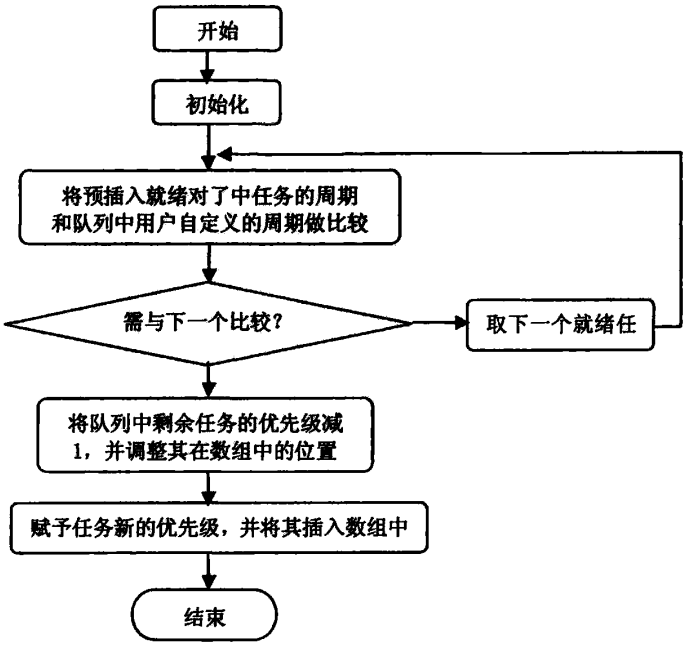

图5-1 单调速率调度算法流程图

Fig. 5-1 Rate Monotonic Schedule Algorithms Flow

在单调速率调度算法中，系统创建的时候需要用户提供任务的周期，因此将OSTaskCreate()和OSTaskCreateExt()中的参数INT8U Prio改为INT8U Period，用来记录任务的周期，在函数内定义局部变量INT8U Prio用来记录分配给任务优先级。增加项目INT8U csRec[64][2]，记录每个任务的周期。在OS_TCB中添加任务标识号TaskID，用来记录任务的周期。在该算法中也要在系统中增加OSTaskPrioCreate()函数，处理创建或删除一个任务的时候处理任务优先级调整。

5.2.2 运行结果

建立一个优先级为16的任务T1、一个优先级为17的任务T2，然后建立三个优先级为18的任务T3、T4、T5。

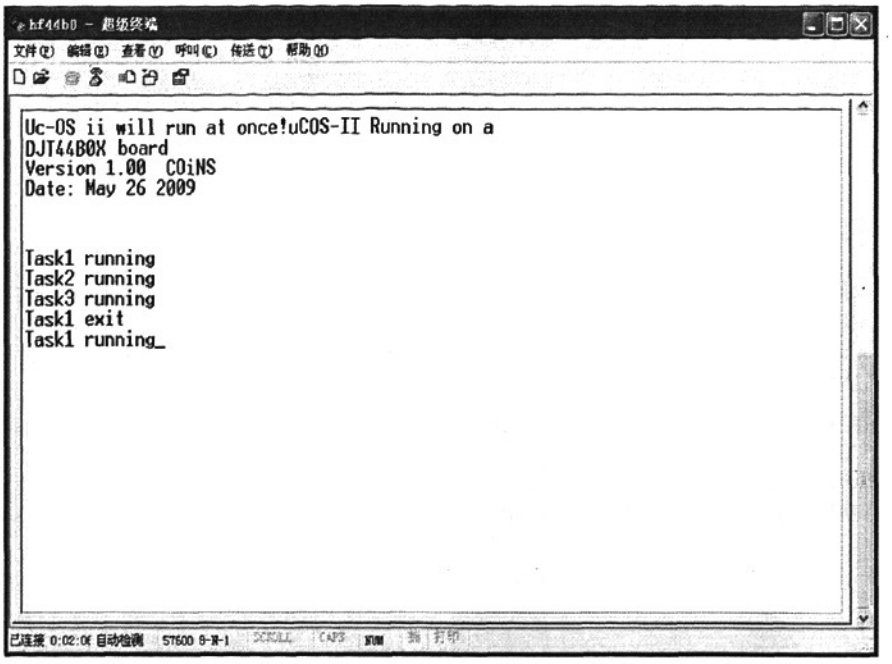

图 5-2 μC/OS-II 上运行相同优先级任务

Fig. 5-2 Same Priority Tasks Run in μC/OS-II

实验结果表明，μC/OS-II不能调度相同优先级任务。因此T4和T5得不到调度。

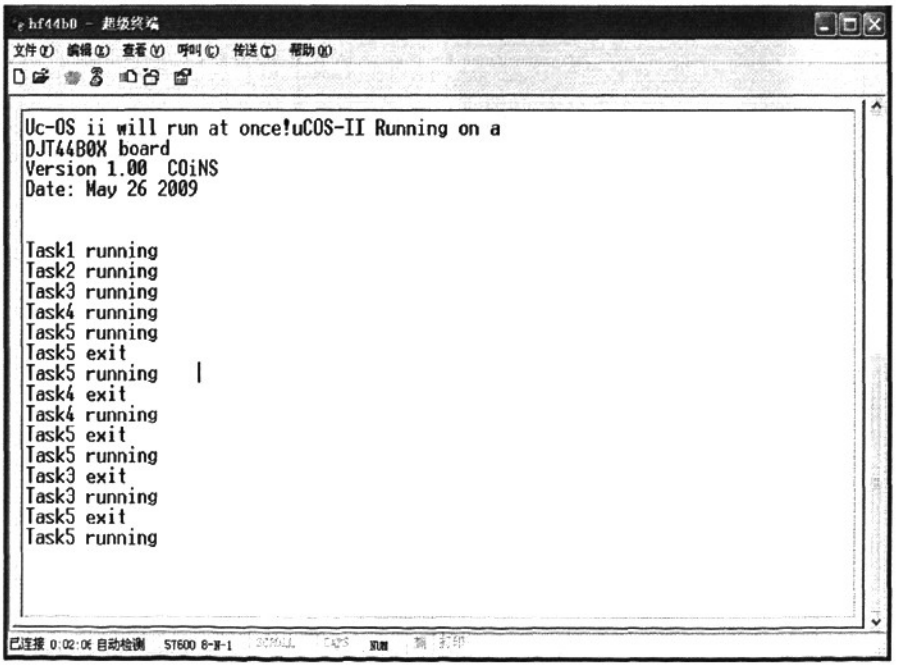

图 5-3 改进后的 μC/OS-II 运行相同优先级任务

Fig. 5-3 Same Priority Tasks Run in Modified μC/OS-II

在改进的内核中运行，运行结果如图 5-3 所示。实验结果表明，改进后的内

核不但可以实现相同优先级任务的调度，还可以有不同优先级任务的调度，相同优先级的任务按照 RM 算法分配的优先级进行调度，并保持了 μC/OS-II 抢占调度的特性。

5.3 μC/OS-II 优先级反转

在调度策略上，μC/OS-II采用的是基于优先级的可抢占式PBP(Priority Based preemptive)调度策略。理想的情况下，为保证实时性能，高优先级任务总是抢占低优先级的任务。但是在有些复杂实时应用中会有多任务共享资源的要求，在这种情况下，低优先级任务拥有高优先级任务所需的共享资源时，优先级反转现象就会发生。此时低优先级任务可能会阻塞高优先级任务直到低优先级任务释放此共享资源。如果低优先级任务被不需要此共享资源的中等优先级任务所抢占，那么优先级反转则会加剧甚至会产生无限反转现象。此时高优先级任务会长期阻塞而得不到运行，从而错过高优先级任务的截止期，导致错误发生甚至引起实时应用系统崩溃。这就是所谓的优先级反转[48](Priority Inversion)现象。

5.3.1 优先级反转问题分析

对于可抢占调度的实时内核，一般高优先级的任务应该能够从初始化开始就抢占较低优先级的任务并立即执行。优先级反转是指一个高优先级任务由于申请的资源被低优先级任务占用而被阻塞的现象。如果这时有中等优先级的就绪任务，阻塞会进一步恶化，甚至超过了任务的截止期而造成严重的后果。这种情况通常产生在两个任务同时要访问共享资源的时候，直接导致了不确定的优先级反转时间和系统可调度性的降低，从而危害了系统的实时性能。

假设T1、T2、T3是实时系统中的三个任务，其优先级依次降低，T1与T3共享一个资源，且在使用共享资源之前，必须首先得到该资源的信号量S。如图5-4示(有阴影填充部分表示使用共享资源)：

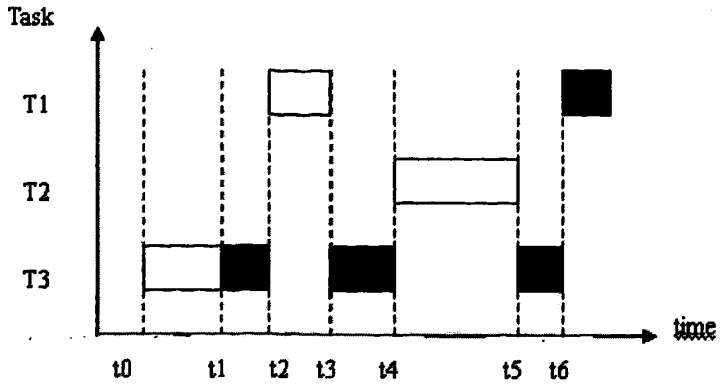

图5-4 优先级反转示意图

Fig.5-4 Scheduling of Priority Inversion

在t0时刻，T3开始运行，而具有较高优先级的T1和T2处于挂起状态，等待某一事件的发生。到了t1时刻，T3得到信号量S并开始使用共享资源。t2时刻，T1等待的事件到来，从而剥夺了T3的CPU使用权，开始执行。t3时刻，TI要使用共享资源，而此时S正被T3占用，T1只好挂起并重新将CPU的使用权交给T3。这时，T1就被T3阻塞了。更糟的是，在t4时刻，T2等待的事件发生，由于其优先级高于T3并且T2不申请该信号量S，因此抢占T3执行直至完毕。然后T3在t5时刻接着运行直到t6时刻释放信号量S。

上述的情况就称为优先级反转，因为此时T1的优先级实际上已经降到了T3的水平。解决RTOS的优先级反转问题，目前典型的实时设计模式有两种：优先级置顶协议[49](priority-ceiling Protocol)和优先级继承协议[50](Priority-inheritance Protocol)。由于μC/OS-II不支持相同优先级的轮转调度从而也就不支持优先级继承协议。μC/OS-II 采用优先级置顶协议来抑制优先级反转。

优先级置顶协议的思想是为每一种共享资源设置一个互斥信号量，同时要为所有共享资源的任务设置一个置顶优先级PCP，该优先级高于所有占有共享该资源任务的优先级。当占有共享该资源的某个任务进入临界区时，立即将该任务的优先级置顶，使其尽快退出临界区，以减轻优先级反转问题。图5.2说明了优先级置顶协议工作方式。

优先级继承协议基本思想是将每一种共享资源设置一个互斥信号量，当共享资源的某个任务进入临界区时，立即将该任务的优先级提升至共享该资源的最高优先级任务的优先级，使其尽快运行并退出临界区。该模式下抑制优先级反转，优先级的提升和恢复可以在程序员非干预下动态自动完成，从而减少开发开销。

也可以使系统很好的适用于复杂应用。

5.3.2 抑制优先级反转的方法

1．在 μC/OS-II 中的调度器加锁

调度器加锁法解决问题的思路是使占有共享资源的低优先级任务尽快的完成，退出临界区。当低优先级任务占用共享资源时，将调度器加锁，即让其他所有任务停止调度当此任务退出临界区时再开启调度器。这是一种强制的方法，使低优先级任务尽快的完成，再恢复调度。本文利用 μC/OS-II 中的调度器加锁函数 OSShedLockO 和调度器开启函数 OSShedUnLock()实现此方法。

该方法的核心思想是将多任务对共享资源的使用串行化。所施加的限制条件非常强硬，虽然可以抑制反转，但影响了多任务并发的程度。

2．在 μC/OS-II 中的优先级置顶协议

μC/OS-II 利用互斥信号量工具，实现了优先级置顶协议。具体的实现方法是：首先要确定共享某临界资源所有任务的最高优先级，再确定系统中没有被占用的并且高于最高优先级任务的置顶优先级别 PCP(Priority ceiling Priority)。当优先级反转出现时，将正在使用共享资源的低优先级任务的优先级提升到 PCP，从而保证占有临界资源的任务尽快运行并尽早退出临界区。用完临界资源后，低优先级任务恢复原来的优先级，从而很好的抑制了优先级的反转。

优先级置顶协议主要是通过 μC/OS-II 的互斥信号量中的获取函数 OSMutexpend()和释放函数 OSMutexpost()实现的。

OSMutexpend()：获取互斥信号量 Mutex 时，如果 Mutex 可用，并且占有信号量任务的优先级不是最高，则提升此任务的优先级至 PCP，使它尽快执行并退出临界区从而释放 Mutex。

OSMutexpost()：当用 OSMutexpost()释放互斥信号量时，只要简单地将当前任务的优先级恢复原值即可。

然而，这种方法有很多的缺点，首先为每个共享资源分配一个 PCP 很繁杂，开发人员必须对每个分配的优先级都要了解清楚，这增大了应用开发难度。其次，分配的 PCP 过多，会使任务切换次数过多而降低整个系统的效率。最后，过多的分配共享资源的 PCP，会使得可建立的任务数变得十分有限(因为 μC/OS-II 中

只允许有64个任务并且每个任务被分配唯一的优先级)，这使得复杂的嵌入式实时应用难以在μC/OS-II上实现。

3．μC/OS-II优先级继承协议实现

优先级继承协议的核心思想是：高优先级任务申请低优先级任务占有的共享资源时，提高低优先级任务的优先级至高优先级任务的优先级水平。μC/OS-II不支持同优先级任务的调度，但通过对其内核的改造使其支持优先级继承协议。实现过程如下：

为了跟踪实行优先级继承协议时各任务优先级的变化，首先要修改事件控制块ECB的数据结构。在ECB的数据结构中增加已获得信号量的任务的优先级的记录项，使得在阻塞发生时方便修改优先级。此改进方案适用于对于所有任务只对一个临界区进行操作，或者这些任务中虽对多个临界区进行操作但这些临界区并不互相嵌套的情况。

实现优先级继承协议主要是对 μC/OS-II 两个内核函数 OSSemPend()和OSSemPost()

OSSemPend()：获取一个信号量时，如果信号量可用，那么让当前任务获得该信号量，并在ECB中记录当前任务的优先级。如果当前信号量不可用，则将当前运行的任务加入到等待任务列表，然后遍历等待队列。如果占有信号量的任务的优先级不是最高，则实行优先级继承协议，把等待此信号量的高优先级任务的优先级赋给它，暂时提高其优先级，使它尽快执行完临界区而释放信号量。如果占有信号量的任务优先级最高，就不会发生优先级反转，不用提高其优先级。

OSSenPost()：当用OSSenPost()释放信号量时，只要简单地将当前任务的优先级恢复原值即可。这样就有效避免了优先级反转而导致低优先级任务被许多中等优先级任务抢占，使得高优先级任务长时间得不到执行这种最严重的情况。

5.4 RAPIDRMA分析

RAPIDRMA即Rapid Rate Monotonic Analysis，一款为即时系统软体开发者加速开发周期,避免重大设计错误的软体。RAPIDRMA内多种分析工具可以让设计者在各种不同的设计环境中测试软体模型,估计各种不同的实现方案对系统性能最佳化的影响[51]。

针对五个任务T_1,T_2,T_3,T_4,T_5，利用RAPIDRMA工具做可调度性分析。

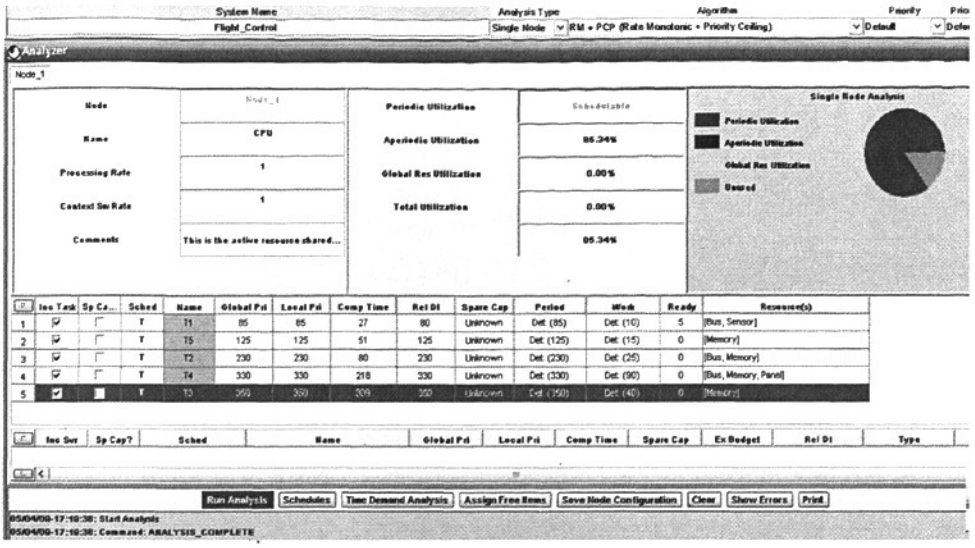

图5-5 任务可调度性分析

Fig. 5-5 Task Schedule Analysis

对于图5-5，按照从左上到右下边的顺序看：分析类型为单节点（single node），指本例中的任务运行在同一个处理器上；调度算法采用的是单调速率+优先级继承协议算法（RM+PCP）（Rate Monotonic + Priority Ceiling）；优先级和优先级映射像采用默认选项，即任务的优先级根据所选择的调度算法来分配和映射。调度结果显示：Schedulable表明这5个任务都是课调度的；周期利用率是5个任务的处理器利用率；图中的饼图是任务对处理器的利用率分布图，包括周期性任务、非周期性任务、资源利用率等（本文只讨论了周期性任务）。图中下半部分的电子表格是对本例中5个任务可调度性分析相关数据的总括：指定参与可调度性分析的任务选项（Inc Task）；针对某一具体任务的可调度性分析结果项（Sched）（T为可调度，F为不可调度）；每个任务的名称项（Name）；优先级项（Local Pri）；最坏情况执行时间（Comp Time）；期限项（Rel Dl）；周期项（Period）；执行时间项（Work）；任务共享资源项（Resources）。

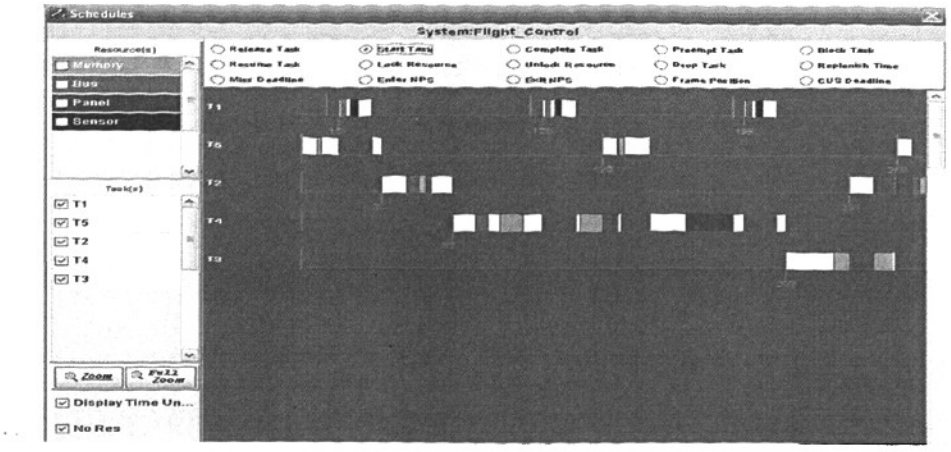

图5-6 调度表

Fig. 5-6 Schedule chart

图5-6是对各个任务调度时刻的反映。图中标注的时刻为任务的开始时刻（Task Start）。该图表明了分析中所包含的周期任务、周期服务和共享资源的最坏调度情况。图中表示了从0时刻到任何具有最坏执行时间任务执行结束，任何共享资源和所选择的周期任务执行情况。每个任务或者资源用其旁边的水平线表示。任务的开始用其时间线上的垂直线表示。当任务运行时时间线会有一个白色的条文，从开始到任务结束或者被中断。

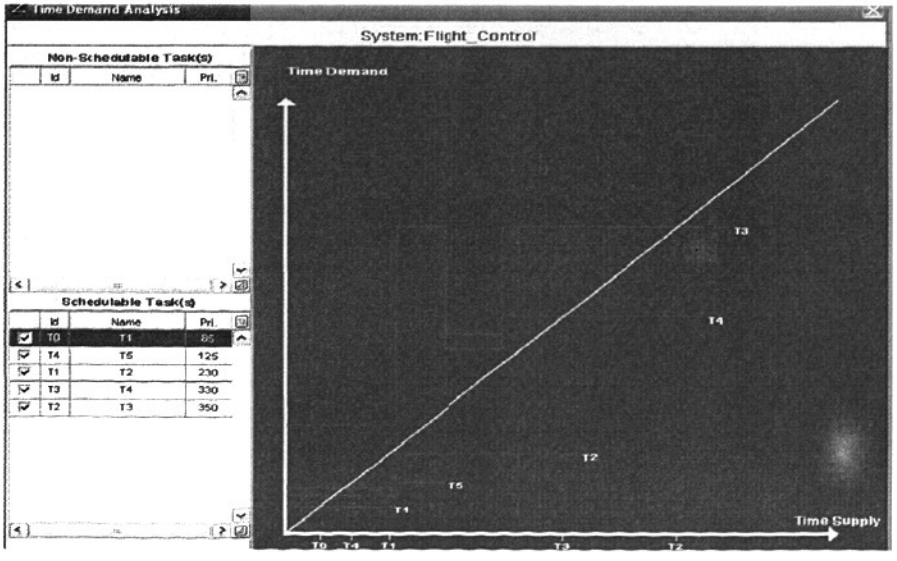

图5-7 需求时刻分析

Fig. 5-7 Time Demand Analysis

图5-7中水平轴（Time Supply时间供给）表示完成时间的仿真值，垂直轴（Time Demand时间需求）表示完成时间的计算值。对角线表示时间供给等于时间需求。

每个任务从0时刻到截止时刻用线段表示，连接了计算的需求时间和每一步的分析的仿真时间。如果任务在截止期之前和对角线相交，表明任务是可调度的。图形的左边显示了可调度任务和不可调度任务。

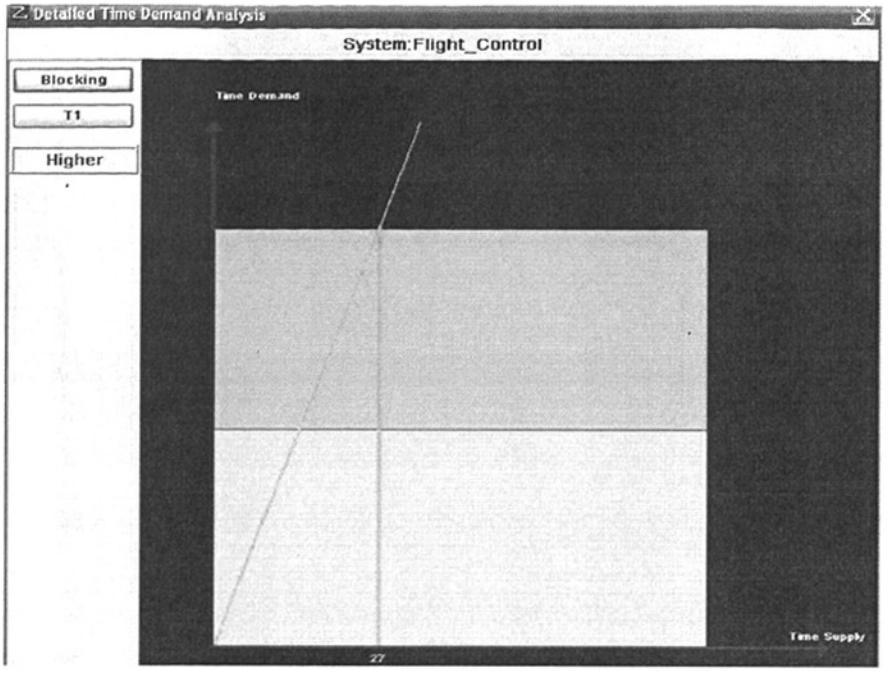

图5-8 任务T1的详细需求时刻分析

Fig. 5-8 Detailed Time Demand Analysis of Task T1

图5-8中水平轴(Time Supply时间供给)表示完成时间的仿真值，垂直轴(Time Demand时间需求）表示完成时间的计算值，图中斜线与垂直线的交点处表示时间供给=时间需求（斜线上下两部分面积相等），垂直线在水平轴对应的数值即是任务的最坏执行时间为27ms，该值小于其期限值80ms，所以T1可以调度。条形图有两个部分组成，下面部分是高优先级的抢占了。低优先级任务阻塞的时间需求用粉红色表示，高优先级挂起的时间部分用绿色表示，任务执行时间用黄色表示。延迟产生的时间用红色表示。如果条形图穿过了对角线，则该任务可调度。

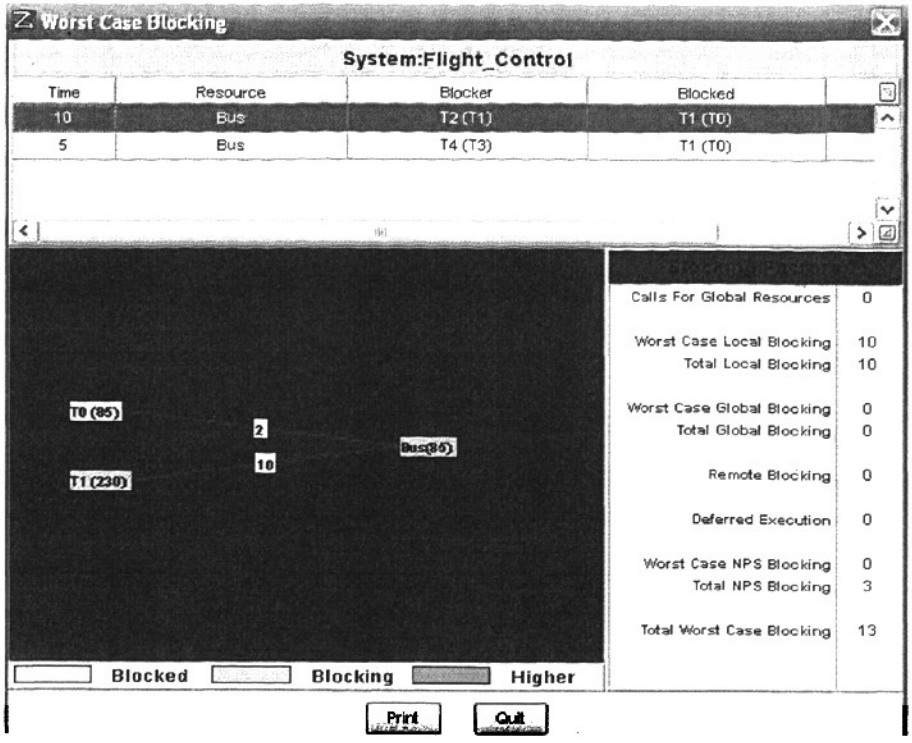

图5-9 任务T1的最坏情况执行时间分析

Fig. 5-9 Worst Case Completion Time Analysis of Task T1

图5-9是对的最坏情况执行时间图形化的明细表。“Worse Case Blocking”图是T1被阻塞的详细情况：表格部分表明T1因和T2、T4共享资源BUS被其阻塞13ms，连线图部分证明T1（ID：T0）被T2（ID：T1）、T4（ID：T3）阻塞时间分别为10ms、5ms，T1和T2共享资源的时间分别为2ms和10ms。“Blocking Factors”表示T1被阻塞的最长时间为13ms。因此可以计算出T1的最坏情况执行时间=被抢占时间+执行时间+被阻塞时间=27ms。

5.5 本章小结

本章主要研究了单调速率调度算法，对其可调度充分条件和充分必要条件进行了研究，并和期限最近者优先调度算法进行比较。对所选择的单调速率调度算法进行了改进，使之可以对同优先级的任务进行调度，又保持了μC/OS-II的抢占式调度。最后应用RAPIDRMA工具对任务的调度结果进行了分析。

结论

本文介绍了实时操作系统的概念、体系结构、衡量标准和实时内核。以μC/OS-Ⅱ实时操作系统为研究对象，研究了其结构、特点和任务管理。任务管理主要研究了任务状态、任务控制块、任务创建和删除、任务调度以及任务切换。在此基础上对 μC/OS-Ⅱ实时内核进行了改造，包括：OS_CPU.H 中需要针对具体处理器的字长重新定义一系列数据类型、OS_CPU.H 中需要设置常量来标志堆栈增长方向、OS_CPU.H 中需要声明用于开关中断和任务切换的宏、OS_CPU_A.ASM 中需要改写 4 个汇编语言的函数、OS_CPU_C.C 中需要用 C 语言写 6 个简单的函数。将其移植到嵌入式微处理器 S3C44B0 中，并对移植进行了测试。

本文阐述了实时系统的调度理论和调度算法的研究现状。重点研究了单调速率调度算法，对其可调度充分条件和充分必要条件进行了研究。进而对 μC/OS-II 实时操作系统内核调度方式进行了分析、研究，提出单调速率调度算法和固定优先级调度算法相结合的混合调度算法。通过对 μC/OS-II 内核的改造，添加进同优先级单调速率调度算法。将现有的 μC/OS-II 的单一固定优先级抢占调度算法与之结合，形成了一种混合的调度策略，可以更加灵活的适应实际的应用。最后用 RAPIDRMA 工具对任务调度进行了分析，并通过运行结果的分析说明了任务的可调度性。

本文阐述了μC/OS-II实时内核中优先级反转问题产生的原因和对系统实时性的影响，提出了调度器加锁及优先级置顶、优先级继承三种抑制μC/OS-II优先级反转的实现方法。

本文涉及到的调度方法中，任务相对简单，主要是针对周期任务调度。而实际应用中任务不一定都是周期性的，对各种不同特征复杂任务的混合调度还需要进一步研究。对于更多任务的系统，可以对内核进行改造以增加可调度的任务数。在改善资源管理方法时，所使用的资源数目相对较少，临界资源嵌套的情况还未讨论，在以后的研究中，将对这种复杂的资源管理进行更深入的研究。

参考文献

[1] 田泽.嵌入式系统开发与应用.北京航空航天大学, 2005.1

[2] Batel,etal. Real-time embedded system. Computing & Control Engineering journal.2002(4):154-156

[3] http://hi.baidu.com/zhenjiahf/blog/item/a39e6a8f78a01af2513d9297.html

[4] Alan Burns, Andy wellings.实时系统与编程语言.王振宇,陈利等译.北京:中国机械出版社,2004.

[5] J A Stankovic.Misconceptions about Real-Time Computing: A Serious Problem For Next-Generation Systems.IEEE Computer, 2003(10):10-19.

[6] 汤子瀛,杨承忠,哲风屏.计算机操作系统.西安:西安电子科技大学出版社,1996

[7] Alan C Shaw. Real-Time Systems and Software.New Youk: John Wiley and Sons, 2001

[8] 郑宗汉.实时系统软件基础北京:清华大学出版社,2003

[9] 王金刚.基于Vxworks的嵌入式实时系统设计.北京:清华大学出版社,2004.

[10] 董晓霞.基于QNX实时操作系统的编程应用.雷达科学与技术,2003(3):34-39.

[11] 陈向群.Windows CE.NET系统分析及实验教程.北京:机械工业出版社,2003.

[12] Jean.J.Labrosse.嵌入式实时操作系统μC/OS-Ⅱ(第2版).邵贝贝等译.北京:北京航空航天大学出版社, 2003.

[13] A S Tancnbaum. Modern Operating System.Upper Saddle River: Prentice Hall, 1999.

[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] 金宏,王强,王宏安等.基于动态抢占阈值的实时调度.计算机研究与发展,2004,41(3):339-398.

[16] J A Stankovic, M Spuri, K Ramamritham, G C Buttazo. Deadline Scheduling for Real-Time Systems-EDF and Related Algorithms. Boston: Kluwer Academic Publishers, 1998.

[17] B.Andersson, J.Jonsson. Fixed-priority preemptive multiprocessor scheduling:

To partition or not to partition. In: Proc of the International Conference on Real-Time Computing Systems and Application, Cheju Island, Korea, 2000: 337-346.

[18] 邢群科,温天江.时钟驱动的原理和实现.微计算机信息,2005,21(7):136-138

[19] 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.

[20] 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.

[21] A K Arekh.A Generalized Processor Sharing Approach to Flow Control in Integrated Services Networks.Massachusetts Institute of Technology, 2001.

[22] Kuo Tei-wei, Yang Wang-ru, Lin Kwei-jay.EGPS: A Class of Real-Time Scheduling Algorithms Based on Processor Sharing. In: Proc of the 10th Euromicro Workshop on Real-Time Systems. Los Alamitos, California: IEEE Coputer Society Press, 1998:27-34.

[23] J P Lehoczky, L Sha, J K Stronsnider. Enganced Aperiodic Responsiveness in Hard Real-Time Environments.In: Proc of the 8th IEEE Real-Time System Symposium. San Jose, California: IEEE Computer Society Press, 2004: 210-217.

[24] 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.

[25] T M Ghazalie, T P Baker. A Periodic Servers in a Deadline Scheduling Evironment [J]. Journal of Real Time System, 2005, 9(1): 31-67.

[26] M Spuri and G Buttazzo.Scheduling Aperiodic Tasks in Dynamic Priority System. Journal of Real-Time Systems, 1996, 10(2): 179-210.

[27] B Sprunt, L Sha, J P Lehoczky.Aperiodic Task Scheduling for Hard Real Time Systems. Journal of Real Time Systems, 2000, 1(1):27-60.

[28] L Abeni, G Buttazo. Integrationg Multimediz Applications in Hard Real Time System.In: Proc of the 19th IEEE Real Time Systems Symposium. Los Alzmitos, IEEE Computer Society Press, 1998: 4-13.

[29] J P Lehoczky,S Ramos-Thuel. An Optimal Algorithm for Scheduling Soft Aperiodic Tasks in Fixed Priority Preemptive Systems.In: Proc of the 13th IEEE Real Time Systems Symposium. Phoenix.Arizona: IEEE Computer Society Press, 2002: 110-123.

[30] Gerhard Fohler. Joint Scheduling of Distributed Complex Periodic and Hard Aperiodic Tasks in Statically Scheduled Systems.In: Proc of the 16th IEEE Real Time Systems Symosium. Pisa, Italy: IEEE Computer Society Press, 2002: 152-161.

[31] Robert Davis, Dual Priority Scheduling.A Means of Providing Flexibility in Hard Real Time Systems. Department of Computer Science, University of York, Tech Rep: YCS-230, 1994: 101-122.

[32] Robert Davis, Andy Wellings, Dual Priority Schedulin.In: Proc of the 16th IEEE Real Time Systems Symposium. Pisa, Italy: IEEE Computer Society Press, 1995: 100-109.

[33] 王济勇,林涛,王金东等.EDF 调度算法抢占行为的研究及其改进.电子学报,2004,32(1): 64-68.

[34] 金宏,王宏安,王强等.改进的最小空闲时间优先调度算法.软件学报,2004,15(8):1117 一 1132.

[35] 张惠娟,周利华.一种基于 EDF 算法的多处理器实时调度算法.计算机工程与应用,2003,30: 16-17.

[36] 魏立峰,于海斌.一种基于自适应控制的软实时调度算法研究.系统仿真学报，2004,16(4):760-771.

[37] http://www.micrium.com

[38] 谢长生,马进德,黄浩.基于 μC/OS-Ⅱ的任务调度策略研究.计算机工程与科学,2004,26(8):70-73.

[39] C L Liu, J W Layland.Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment.Journal of the Association for Computing Machinery,1973.1,Vol.20(1):46-61.

[41] 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,Vol.15(6):799-814

[42] Liu Sha, Ragunathan Rajkumar. Priority Inheritance Protocols: An Approach to Real-time Synchronization. IEEE Transactions on Computers. 1990.9, Vol. 39(9):1175-1185

[43] 翟鸿鸣.单处理器系统的实时调度算法研究.微机发展,2003,l3(10):99-110.

[44] M.H.Klein, J.P.Lehoczky,etc. Rate-Monotonic Analysis for Real-Time Industrial Computing. IEEE Computer. 1994.1: 24-33

[45] Paulo Martins. Integrating Real-Time UML Models with Schedulability Analysis. Tri-Pacific Software Inc. White Paper. http://www.tripac.com

[46] Wayne Wolf. 嵌入式计算机系统设计原理. 孙玉芳，梁彬等译. 机械工业出版社,2002

[47] 邹勇.开放式实时系统的调度方法研究.中国科学院博士学位论文. 2003.5

[48] D.Kalinsky and M.Barr. Introduction to priority inversion. http://www.embedded.com

[49] J.B.Goodenough and L.Sha. The priority ceiling protocol: A method for minimizing the blocking of high priority Ada tasks. ACM SIGAda Ada Lett,1998,8(7):20-31

[50] L.Sha, R.Rajkumar, J.P.Lehoczky. Priority Inheritance Protocols: An Approach to Real-Time Synchronization. IEEE Transactions on computers, 1999,39(9):1175-1185

[51] www.tripac.com

攻读硕士学位期间发表的论文

[1] 张波[1].USB 数据采集在温度控制系统中的应用,扬州大学学报(自然科学版) 2008-07.

[2] 张波[1]、邓则名[2].直流调速系统的 SIMULINK 仿真, 电子测试, 2008-06.

[3] 张波[1]、张维威[2].图像处理在粘度测量中的应用, 电脑与电信, 2008-08.

[4] 张波[1]、郑鹏[2].轴类零件形状误差优化评定的研究, 机电工程, 2008-08.

[5] 张波[1].电气专业英语教学浅谈, 广东海洋大学学报, 2008-08.

致谢

在论文完成之际，向所有帮助我的良师益友和家人朋友致以最真诚的谢意。

首先感谢我的导师邓则名教授和高军礼副教授。整个的研究工作和论文撰写工作都是在邓老师和高老师的悉心指导下完成的。在开题时，他帮助我们仔细选择课题、确定方案；在研究工作进行的时候他给我们悉心指导，并尽力提供各种条件；论文撰写的时候，他时时关注论文的进展，并给出指导性意见；最后对论文进行了多次的审阅和细致的修改。三年来，邓老师和高老师在学习研究上给予我们指导和支持，在生活上给予我们关心和爱护。邓老师和高老师渊博的知识、敏锐的洞察力、开阔的思路、严谨的治学态度、尽职尽责的工作精神、宽厚仁慈的性格以及高尚的人格魅力都给我留下了深刻的印象，使我受益匪浅。

感谢我的工作单位广东海洋大学的同事们，在我读书期间在工作中给与学习和生活上的关怀，帮我分担了不少的工作任务。

感谢同学杨土权、张维威、康亚芬、康慧、佟向坤、杨刚，师弟梁永忠、陈世军、权宁辉、刘汉东、张英范和同实验室其他的同学的帮助。

更要深深感谢的是我的父母、姐姐和弟弟，在精神上对我的鼓励和支持。感谢我的男朋友万宗臣一直以来的默默支持和鼓励。他们的陪伴，让我顺利渡过了三年的学习生活。