南京航空航天大学

硕士学位论文

基于多核的嵌入式操作系统的研究和设计

姓名：王凯

申请学位级别：硕士

专业：计算机科学与技术

指导教师：徐涛

2010-12

摘 要

随着计算机技术的发展，嵌入式实时系统在众多领域得到广泛应用。相比于单核处理器，多核处理器能够使嵌入式系统获得更高的性能。在 PC 全面进入多核时代的背景下，嵌入式领域的多核技术也逐渐成为热点，而支持 SMP（对称多处理）的嵌入式操作系统，由于其共享内存、负载均衡、性能功耗比高等优点，能够充分发挥多核处理器的优势。μC/OS-II 是一个源码公开、微内核的嵌入式实时操作系统(RTOS)，但是它不支持多核处理器，因此本文主要对 μC/OS-II 扩展到多核处理器的关键技术进行研究。

本文首先分析了 RTOS 主要的性能指标，接着分析了嵌入式 Linux 对 SMP 的支持，然后分析了 μC/OS-II 内核的结构和性能，对 μC/OS-II 只有 64 个优先级的缺点提出了两种改进算法：优先级总量扩充算法和同优先级调度算法，为 μC/OS-II 扩展到多核处理器提供了重要的参考。

本文的主要工作是在重用 μC/OS-II 模块的基础上，设计了支持多核处理器的 M_μCOS 微内核。首先提出了 M_μCOS 具体的设计目标并完成了建模工作，接着本文重点分析并解决了多核的启动顺序、多核间的互斥、任务管理等关键问题：在多核的启动顺序问题上，提出了启动核(BP)和应用核(AP)的分工，让 BP 负责全局变量和内存的初始化工作；针对多核的互斥问题运用了自旋锁机制，分析了经典自旋锁算法的不足并改进了自旋锁；在任务管理问题上，针对于 M_μCOS 内核的特点设计了并行调度模型，并给出了多核任务调度算法。

经过实验证明，本文所设计的基于多核处理器的嵌入式实时操作系统 M_μCOS 能够成功移植到由 NiosII 软核所构建的多核处理器平台上。

关键词：μC/OS-II，多核处理器，微内核，调度算法，移植

ABSTRACT

With the development of computer technology, embedded real-time system is widely used in many fields. Compared with single-core processor, multi-core processors enable embedded real-time system to achieve higher performance. When PC entered into the multi-core era, embedded multi-core technology gradually hot. Embedded operating system which support SMP (symmetric multi-processing) can make better use of multi-core processors because of its sharing memory, loading balance and high performance. However, many Embedded Real-Time Operating Systems(RTOS) do not support multi-core processors such as μC/OS-II studied in this paper which is an open source, micro-kernel embedded real-time operating system. Therefore the main purpose of this paper is to study the key technologies to supports multi-core processors on SMP structure for μC/OS-II.

This paper analyzes the performance of RTOS and describes the support of SMP in embedded Linux operating system. Then analysis the structure and work process of each module of the μC/OS-II kernel, give two solutions for the problem of insufficient processing tasks: expansion of the total priority and same priority scheduling, providing a good preparing for μC/OS-II using in multi-core environment.

Main work of this paper is to design a micro-kernel M_μCOS which support embedded multi-core processors according to reuse the model of μC/OS-II. First proposed a specific design goal for M_μCOS, then finish the modeling work. Focuse on start boot sequence, mutual exclusion mechanism and task scheduling algorithm in the multi-core environment. Proposed division of work for the booting Processor(BP) and the Application Processor(AP), and BP is charge for global variables and memory initialization. Spin-lock mechanism is proposed and improved. Finally design the task scheduling model and scheduling algorithm for multi-core environment.

The experiments show: M_μCOS can successfully transplante to the multi-core processors platform building by soft core NiosII .

Keywords: μC/OS-II, multi-core processors, micro-kernel, scheduling algorithm, porting

图表清单

注释表

略写	英文全称	中文名称
SMP	Symmetric Multi-Processing	对称多处理
AMP	Asymmetric Multi-Processing	非对称多处理
RISC	Reduced Instruction Set Computer	精简指令系统(集)计算机
RTOS	Real-Time Operating Systems	实时操作系统
DSP	Digital Signal Processing	数字信号处理
MMU	Memory Management Unit	内存管理单元
DMA	Direct Memory Access	动态内存存取
MIPS	Million Instructions Per Second	每秒处理的百万级的机器语言指令数
APIC	Advanced Programmable Interrupt Controllers	高级可编程控制器
MPP	Massively Parallel MutiProcessing	大型并行处理系统
UML	Unified Modeling Language	统一建模语言
ISR	Interrupt Service Routines	中断服务程序
TCB	Task Control Block	任务控制块
ECB	Event Control Blocks	事件控制块
API	Application Programming Interface	应用程序编程接口

承诺书

本人声明所呈交的硕士学位论文是本人在导师指导下进行的研究工作及取得的研究成果。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，也不包含为获得南京航空航天大学或其他教育机构的学位或证书而使用过的材料。

本人授权南京航空航天大学可以将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、汇编学位论文。

（保密的学位论文在解密后适用本承诺书）

作者签名：_____

日　　期：_____

第一章 绪论

1.1 引言

随着科学文明的不断进步，信息技术在推动人类生产力发展中起到越来越占决定性的作用。作为信息技术的重要组成部分，计算机技术经历了惊人的发展过程。从第一台计算机产生至今短短几十年，人类的生产方式和生活方式产生了翻天覆地的变化，从而推动了人类社会的更大进步。计算机技术不断进步的动力近年来主要体现在以并行技术为代表的计算机体系结构的发展[1]。目前大部分处理器厂商都致力于采用新的体系结构技术来提高计算机的性能，从而导致了各种多核处理器的产生。多核处理器[2]就是将两个以上的处理器核封装在一个芯片内，在相同的主频下获得更好的性能。

嵌入式系统是一种对功能、可靠性、成本等有严格要求的专用计算机系统[3]，主要具有软硬件可裁减的特点，在各行各业中的应用比较广泛。随着嵌入式系统应用的不断扩展，嵌入式已成为工业化职能化的必由之路，嵌入式产品已经深入到各行各业中。在今天，嵌入式操作系统以其广泛分布各领域，技术比较成熟、使用嵌入式操作系统，可以大大缩短产品开发周期，降低产品成本，提升产品性能。

嵌入式系统的应用主要包括[4]：国防武器装备中的导弹瞄准器，电子对抗设备等、通信系统中的路由器，移动电话等、工业控制中的各种控制器、智能仪器和各种各样的家电产品，可以说嵌入式应用前景广阔，无处不在。嵌入式系统的组成如图1.1所示，它一般由嵌入式微处理器、外围设备、嵌入式操作系统及嵌入式应用软件等四个部分组成，通过应用程序管理和控制其它设备，嵌入式操作系统是嵌入式系统的核心，起到连接应用程序和硬件系统的纽带作用。

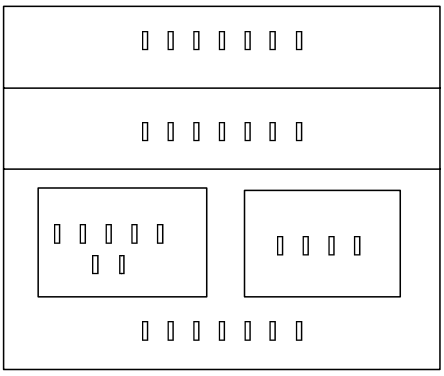

图 1.1 嵌入式系统的组成

嵌入式系统是以应用为中心，以计算机技术为基础，且软硬件可裁剪，适应应用系统对功

能、可靠性、成本、体积、功耗有严格要求的专用计算机系统。嵌入式系统的特点概括如下：

(1) 系统内核一般都比较小。嵌入式系统一般用于小型的电子设备，需要管理的资源有限，因此内核中相应的功能就简化或者取消了。

(2) 专用性。嵌入式系统的专用化和个性化很强，软硬件系统的结合比较多，对于不同的硬件需要修改和移植相应的软件，即使是同一个系列的产品也是如此。对于应用场合不同的任务需要进行较大的修改。

(3) 系统精简。嵌入式系统一般不区分系统软件和应用软件，没有复杂的设计和实现过程，有时候只是一个简单的任务循环，这样做的目的一方面实现了对成本上的控制，另一方面有利于系统安全。

(4) 高实时性的操作系统。实时性是嵌入式软件的基本要求，在工业控制中必须保证系统的实时性，因此嵌入式操作系统一般要求高实时性。

(5) 嵌入式系统开发需要专门的开发工具和环境。

1.2 研究背景

早期的嵌入式系统因为应用比较简单，是不需要操作系统的，只需要设计一个单任务的循环控制。随着嵌入式系统性能的增长，操作系统显得越来越重要，从而发展为嵌入式操作系统(Embedded Operating System)。嵌入式操作系统以其广泛分布各领域，技术比较成熟、使用嵌入式操作系统，可以大大缩短产品开发周期，降低产品成本，提升产品性能。

嵌入式操作系统具有体积小、实时性、可根据需求裁减内核等特点而受到欢迎，目前市场上的商用嵌入式操作系统如Vxwork[5]、PSOS等已经十分成熟，提供有力的开发和调试工具，但开发成本昂贵。μC/OS-II是基于优先级的抢占式实时多任务操作系统, 包含了实时内核、任务管理、时间管理、任务间通信同步和内存管理等功能，绝大部分代码用C语言写成，与硬件相关部分用汇编语言编写，且它的源代码是公开免费的，uC/OS-Ⅱ是面向中小型嵌入式系统的。包含全部功能模块的内核大约为10KB，如果经过裁减只保留核心代码，则可压缩到3KB左右，所以对其内核进行更改是可以实现的。

随着嵌入式系统应用的不断扩展，嵌入式已成为工业化职能化的必由之路，嵌入式产品已经深入到各行各业中。一方面，嵌入式系统应用领域的不断拓展，对嵌入式系统计算能力都有了更高的要求，致使过去常常用于嵌入式开发的单处理器系统已经无法满足需求。另外一方面，随着芯片技术的不断发展，采用多处理器协同工作来提高嵌入式系统的计算能力，降低响应时间已成为一种必然趋势。嵌入式系统的并行化直接要求嵌入式操作系统对多处理器的支持，因此对嵌入式系统的并行化研究，尤其是对通用的基于单核的嵌入式操作系统对于多处理器环境下的应用和改变显得尤为关键。多核处理器较传统的单核处理器有着高性能、低功耗、易于编

程的诸多优势，因此，多核技术已成为嵌入式系统领域研究和应用的热点。

单处理器速度的不断提升，导致了其性能上出现了瓶颈，代价与处理器速度的比值越来越大，急需从技术上进行新的设计，因此采用并行处理将任务进行分解是一种比较理想的解决办法。多核的技术思想是：将两个或者多个独立的核封装到一个芯片内部，由多个核并行地计算。通过这样做，不仅获得较高处理器性能，还可以将处理器主频保持在较低的水平。虽然目前主流的操作系统已经对PC上的多核处理器提供了完善的支持，但在嵌入式领域，多核处理器的应用很少。传统的实时操作系统还没有完全提供对多核处理器的支持，如本文所研究的uC/OS-Ⅱ是不支持多核处理器的实时操作系统。

操作系统是计算机系统的资源管理者。操作系统的重要任务之一是有序地管理计算机中的硬件、软件资源，跟踪资源使用状况，满足用户对资源的需求，协调各程序对资源的使用冲突，为用户提供简单、有效的资源使用方法，最大限度地实现各类资源的共享，提高资源利用率，从而使得计算机系统的效率有很大提高。因为所研究的uC/OS-Ⅱ是一个主要提供任务管理等基本功能的微内核，所以本文主要从任务管理的角度来研究基于多核的实现方案。

1.3 国内外现状和研究意义

20 世纪 80 年代，商业化的嵌入式操作系统蓬勃发展，目前国内外已有几十种商业操作系统可选，如 Vxworks、pSOS、OS、Windows、RTEMS[7]等。在中国，嵌入式操作系统分为两大类：一类是国内企业开发拥有自主产权的操作系统，如 Hopen OS(女娲)和 DeltaOS 等[8]；另一类是基于 Linux 的操作系统。由于电子技术的快速发展，芯片功能越来越强大，硬件的性能得到进一步增强，也要求嵌入式操作系统与硬件结合，适应于多核环境下更高的要求。本文在准备阶段通过阅读参考文献，掌握了一些主流的嵌入时操作系统的性能和特点。

(1) VxWorks

VxWorks[9]是美国风河系统公司(Wind River)开发的符合工业标准的嵌入式实时操作系统，广泛应用于通信、军事、航空航天等高端技术和实时性要求很高的领域。其特点如下：内核可裁减；256 个任务优先级的多任务系统；共享内存技术；支持信号量机制；支持多处理器并行技术；符合 ANSI C 标准等。VxWorks 的板级支持包非常全面，包括了开发人员的一切支持。但是 VxWorks 是商用嵌入式操作系统，需要购买 Licence 才能使用。

(2) RTEMS

RTEMS 是开源的实时嵌入式操作系统，最早应用于国防系统，由美国 OAR 公司负责升级和维护，在军事领域有着广泛应用。它的特点包括：支持硬实时和软实时的可抢占内核；高度的可裁减性，最小内核 30k，占用系统资源少；核心代码用 C/C++编写，移植性好；支持文件系统；提供多处理器管理等。RTEMS 的性能在某些方面可以 VxWorks 相提并论，它们一般用

于比较高端平台的嵌入式应用。

(3) uClinux

随着 Linux 的快速发展，嵌入式 Linux 发展出很多版本，包括强实时性的 RT-Linux 和一般实时性的 uClinux。其中具有代表性的 uClinux 的特点如下：内核经过裁减后高度优化；保留了 Linux 中常用的 API 函数供用户使用；有完整的 TCP/IP 栈，并支持其他的网络协议栈；支持文件系统；采用实存储器管理策略，通过地址总线直接访问内存，专门针对于没有 MMU(内存管理单元)的处理器。uClinux 适用于低端平台的嵌入式应用，uClinux 的缺点是实时性不高。

(4) μC/OS-II

μC/OS-II 是由美国 Labrosse 先生开发的小型嵌入式操作系统，提供了嵌入式系统的基本功能，其核心代码量较小。μC/OS-II 是基于优先级的抢占式实时多任务操作系统，包含了实时内核、任务管理、时间管理、任务间通信同步和内存管理等功能。绝大部分代码用 C 语言写成，与硬件相关部分用汇编语言编写，且它的源代码是公开免费的。μC/OS-II 是面向中小型嵌入式系统的。包含全部功能模块的内核大约为 10KB，如果经过裁减只保留核心代码，则可压缩到 3KB 左右。μC/OS-II 的特点主要包括：源码公开、可移植性强、可裁剪、稳定性与可靠性都很强。它已经被移植到许多不同种类的 CPU 上，如 ARM 系列、Intel 公司的 80x86 系列、摩托罗拉公司的 PowerPC 系列。表 1.1 列出各实时操作系统性能对照。

表 1.1 实时操作系统性能

性能 / 操作系统	调度算法	任务数量	源码公开	实时性
VxWorks	优先级抢占	256	否	高
RTEMS	优先级抢占	256	是	高
uClinux	时间片轮转	/	是	低
μC/OS-II	优先级抢占	64	是	高

由于多核技术刚刚兴起，在嵌入式应用上还处于研究阶段，一般的嵌入式操作系统并不支持多核处理器，只有少数商用化很高的实时操作系统才提供对其支持，如嵌入式 Linux[11][12]和 VxWorks[13]支持对称多处理(SMP)，Redhat 公司的 eCos 可以在选定的处理器平台上提供 SMP，硬实时操作系统 ThreadX 支持 ARM 公司的 Mpcore[14][15]。本文所研究的微型的源码公开的嵌入式操作系统μC/OS-II 并不支持多核技术，这对我们研究多核架构的嵌入式操作系统造成了阻碍。

μC/OS-II [16]-[18]近年来由于其易移植性、高实时性、可裁减等优点得到了日益广泛的应用，但相比于其它商用嵌入式操作系统，其在任务调度机制中还存在着一些不足。如果能对其进行有效的改进，相信它在嵌入式开发和研究中一定可以获得更加大关注。目前国内外有很多关于 μC/OS-II 内核分析方面的研究，但大多是从 μC/OS-II 的实时性方面入手，提出新的算法对其性

能加以改进，很少有应用于多核技术背景下的研究。

大量的文献资料及从国内外的研究工作表明嵌入式操作系统适用于多核处理器环境的紧迫性和必然性。本文通过分析其它商用嵌入式系统对多核支持所做的工作，更好地指导了 μC/OS-II 下的多核扩展工作。因此如何有效地将操作系统扩展到多核环境，并减少付出代价是非常具有挑战性和研究性的课题。

1.4 论文的目标和组织结构

为了扩展嵌入式实时操作系统，使其能够支持多核处理器，必须对内核进行必要的设计和修改。本文主要从两方面开展工作：一方面分析其它嵌入式操作系统对多核的支持，并结合软件工程思想使用统一建模语言 UML 进行 M_μCOS(区分于改进前的 μC/OS-II)内核的设计；另外一方面分析 μC/OS-II 实时内核的源码及其特点，适当地对其改进，使得 M_μCOS 内核能够移植到多个处理器核心的嵌入式系统中，并对改进后的内核的性能进行测试。

本文的组织结构安排如下：

第一章 绪论部分主要介绍了论文的选题意义和研究目的、国内外的研究现状、论文的所做的主要工作和组织结构。

第二章 主要是嵌入式操作系统的介绍，重点介绍了 μC/OS-II 的微内核的结构和性能，并研究了其调度算法，提出了解决任务优先级不足的改进方式，并提出了支持同优先级调度的改进方式，为微内核的设计和改进做好准备。

第三章 介绍了嵌入式系统的体系结构，从硬件结构、指令集、操作系统等方面介绍了嵌入式系统的结构，简单介绍了对称多处理系统 SMP 的结构和多处理器调度算法的设计要点，主要分析 Linux 系统在内核启动、任务调度算法和资源管理方式等方面对 SMP 的支持，对支持 SMP 的 μC/OS-II 的微内核的设计起到了借鉴作用。

第四章 重点介绍 M_μCOS 内核的设计工作。在设计工作开始前，首先提出了 M_μCOS 内核设计目标作为具体需求，然后对其进行 UML 建模，用例图、类图、状态图等直观表示 M_μCOS 内核的结构和功能，并对微内核的启动顺序、并行调度算法和并行资源管理等关键技术进行研究和分析，提出合理的解决方法。

第五章 介绍了 M_μCOS 移植工作的硬件平台 NiosII 多核处理器及移植过程，并通过实验验证了内核设计工作的正确性。

第六章 工作总结。系统的总结了本文所做工作，并给出未来工作的方向和目标。

第二章 嵌入式系统及其对多核的支持

2.1 嵌入式硬件系统的组成

嵌入式硬件系统以嵌入式微处理器为核心，主要包括嵌入式微处理器、总线、存储器、输入/输出设备等组成部分。和大多数的通用微处理器一样，嵌入式微处理器均采用冯·诺依曼(Von Neumann)结构，它将指令及数据放于同一存储空间中，统一进行编址，指令及数据通过同一条总线访问。

嵌入式微处理器有很多不同的体系，如 ARM、MIPS、PowerPC、X86 等，其中以 ARM 公司（Advanced RISC Machines）生产的 ARM 系列芯片最为著名[19]，它采用精简指令集系统 RISC（Reduced Instruction Set Computer），大部分指令都是单周期指令，通过软件完成部分硬件功能，芯片成本较低。然而即使在同一体系中，嵌入式微处理器也可能具有不同的时钟速度和总线数据宽度、集成不同的外部接口和设备。据统计，目前全世界有几十种嵌入式微处理器体系，嵌入式微处理器的品种总量已经超过一千种。嵌入式微处理器种类很多，按位宽(微处理器一次执行指令的数据带宽)可以分为 4 位、8 位、16 位、32 位和 64 位，按用途可以分为专用于信号处理的嵌入式 DSP 和通用嵌入式微处理器。如表 2.1 数据所示，嵌入式处理器的发展是快速的，系统的性能不断加强。

表 2.1 嵌入式微处理器的发展过程

时期 / 性能指标	80 年代后期	90 年代初期	90 年代后期	21 世纪初
制作工艺	0.8 - 1 μm	0.5 - 0.8 μm	0.35 – 0.5 μm	0.13 - 0.25 μm
主频	< 33 MHz	<100 MHz	<200 MHz	< 600 MHz
晶体管个数	>500K	>2M	>5M	>22M
位数	8/16bit	8/16/32bit	8/16/32bit	8/16/32/64bit

2.1.1 嵌入式微处理器

嵌入式微处理器与通用处理器相比，具有体积小、重量轻、成本低、功耗低的特点，其在工作温度、抗干扰性、可靠性等方面得到加强，嵌入式微处理器除了集成 CPU 核心、Cache、MMU、总线等部分外，还集成了各种外部接口和设备，如动态内存存取 DMA、中断控制器、定时器等。大多数嵌入式微处理器都使用定点运算功能，数值被表示为整数和分数的形式，节

省芯片的成本，当其使用浮点运算时，可以使用软件模拟的方法实现，缺点是这样会占用相应的处理器时间。

为了满足应用的需要，嵌入式微处理器一般可以适当裁减或增加自己的指令集来实现数字处理功能，这些指令主要有：MAC 操作（在一个周期内实现乘法和加法运算各一次）、单指令多数据流操作（一条指令可以实现多个并行数据流计算）、循环指令（通过硬件方式减少循环操作的开销）。

大多的嵌入式微处理系统都有功耗上的限制，可以通过降低工作电压、通过软件设置不同的时钟频率以及关闭暂时不使用的功能模块等方法限制处理器的功耗。比如某个功能模块在设定的时钟周期内没有工作，就关闭它以降低功耗。在功耗管理上，嵌入式微处理器设有不同的管理模式：当有任务运行时处于运行模式；当处理器不执行指令时处于待命模式，此时存储的信息是可用的；当时钟完全停止时处于时钟关闭模式，如果要进入运行模式则需要重新启动微处理器。下面简单介绍几款著名的嵌入式微处理器的特点。

(1) ARM[20]

作为 ARM 型处理器的设计供应商，ARM 公司是微处理器行业知名企业，总部位于英国。ARM 公司设计了大量高性能、廉价、耗能低的 RISC 处理器。ARM 芯片具有性能高、成本低和能耗省的特点。适用于多种领域，比如嵌入式控制、DSP、多媒体应用、手机芯片应用等。许多一流的芯片厂商都是 ARM 公司的授权用户，如 Intel、Samsung、Motorola 等，ARM 已成为业界公认的嵌入式微处理器标准。ARM 处理器共有 7 种不同的处理器模式，表 2.2 为 ARM 的处理器模式。

表 2.2 ARM 的处理器模式

处理器的模式	描述	备注
User	普通程序执行的模式	不能直接切换到其它模式
FRQ	用于高速数据传输或通道处理	快中断响应进入此模式
IRQ	用于通用中断处理	通用中断响应进入此模式
Supervisor	操作系统的保护模式	系统复位和软件中断响应进入此模式
Abort	用于实现虚存或存储保护	无
Underfined	支持软件模拟或硬件协处理器	未定义指令异常响应时进入此模式
System	运行特权操作系统任务	和 User 类似，但能切换到其它模式

1985 年，第一个 ARM 原型在英国剑桥的 Acorn 计算机有限公司诞生，由美国加州 SanJoseVLSI 技术公司制造。 20 世纪 80 年代后期，ARM 很快开发成 Acorn 的台式机产品，形成英国的计算机教育基础。20 世纪 90 年代，ARM 32 位嵌入式 RISC(Reduced lnstruction Set Computer)处理器扩展到世界范围，占据了低功耗、低成本和高性能的嵌入式系统应用领域的领

先地位。ARM 的发展经历了很多版本，从目前支持的最低的 ARM V4 版本一直到可以集成 1-4 个 ARM11 处理器的 ARM Mpcore 版本，把任务进行并行分解，很好地解决了性能和代价之间的矛盾。ARM Mpcore 使用最新的 ARM V7 版本定义，提供了 3 种不同的处理器配置，分别针对不同的应用，满足了下一代嵌入式控制的需求。

(2) MIPS

MIPS（Microprocessor without interlocked piped stages）是一种处理器内核标准，即无内部互锁流水级的微处理器，由 MIPS 公司开发，该公司在 RISC 处理器方面占有重要地位。MIPS 公司既开发 MIPS 结构的处理器，又生产 64 位的芯片。为了方便用户使用其 MIPS 处理器，该公司推出了 MIPS IDF(Integrated Development Framework)开发工具，可以用于嵌入式系统方面的开发。

(3) X86

X86 是最著名的处理器系列，它起源于 Intel 架构的 8080，再发展为 286、386 等，一直到最新的 Pemtium、酷睿系列。在嵌入式市场，8080 是第一款比较主流的嵌入式微处理器。现在的 Pentium 和当初的 8080 使用相同的指令集，保持了很好的兼容性。

(4) NiosII[21]-[24]

NiosII 系列处理器包括 3 种产品：Nios II/f(快速)、Nios II/s(标准)和 Nios II/e(经济)。它们都使用 32 位的指令结构，兼容性好。NiosII 是 Altera 公司开发的软核嵌入式处理器，性能超过 200DMIPS。NiosII 最大的特点是允许用户构建完整的可编程系统，用户可以根据需求添加处理器、存储器、外设等来满足设计和成本的限制。

SOPC Builder[25][26]是 Altera 开发的系统级开发工具，用户可以使用它来快速的配置双核系统，并且可以根据需要选择不同的双核系统类型：资源共享型和不共享资源型。正是由于这些特点，本文最后选择了 NiosII 作为开发平台。

2.1.2 嵌入式系统的总线

嵌入式系统的总线集成在嵌入式微处理器上。从微处理器的角度来看，可将总线分为片内总线和片外总线，其中片内总线连接 CPU 内部各主要部件，片外总线负责 CPU 和外设之间交换信息;从功能和信号类型来看,可将总线分为数据总线 Dbus、地址总线 Abus 和控制总线 Cbus。总线的主要参数包括：总线宽度、总线频率和总线带宽。总线宽度指总线能同时传送数据的位数；总线的频率指总线工作的速度，频率越高速度越快；总线带宽可以由总线宽度和总线频率计算，总线带宽 =(总线宽度/8) * 总线频率。

如图 2.1 所示，片内总线连接了微处理器、Cache 等各个功能部件，而外设通过片外总线与微处理器和高速缓存等相连接。

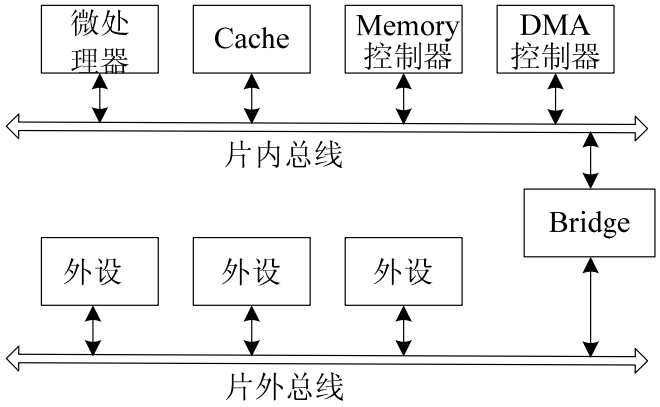

图 2.1 嵌入式系统的总线结构

2.1.3 嵌入式系统的存储器

嵌入式系统的存储器结构是由主存和外存组成的，系统中的代码以及数据一般存储在处理器的主存中。主存是处理器可以直接访问的存储器，系统上电后在主存中的代码可以直接运行，主存具有速度快的特点，一般采用 ROM、EPROM 等。外存是嵌入式处理器不能直接访问的存储器，用来存放各种信息，具有价格低、容量大的特点。

高速缓冲存储器 Cache 是主存的一部分，Cache 中的代码和数据在处理器读取主存时被使用的最多的部分，是主存中相关代码和数据的一个拷贝。Cache 一般都集成在微处理器内部，CPU 每次读取主存时，Cache 控制器都要判断 CPU 要读数据是否在 Cache 中，如果在就称为 Cache 命中。如果要读取的数据不在 Cache 中，则从主存中读取相应数据，并将读取的数据放入 Cache 中，这样做的目的是提高读取主存的速度。

2.1.4 嵌入式系统的指令

指令集是计算机硬件的语言系统，也叫机器语言，它是软件和硬件的主要界面，从系统结构的角度看，是系统程序员看到的计算机的主要属性。以 ARM 处理器指令系统为例，简单介绍 ARM 指令集的类型。ARM 指令的基本格式为：<opcode> {<cond>} {S} <Rd>, <Rn> { , <operand2> }。

<>内为必填项，{}内为选填项；opcode 为指令助记符，如 LDR，STR 等；Cond 为执行条件，如 EQ，NE 等；S 为是否影响 CPSR(current program status register)寄存器的值；Rd 为目标寄存器；Rn 为第一个操作数的寄存器；operand2 为第二个操作数。

ARM 指令集主要包括如下几种类型：

(1) 存储器访问指令：ARM 处理器对存储器的访问只能通过 LDR 和 STR 指令来实现，LDR 是加载指令，STR 是存储指令；

(2) 数据处理指令：大致分为 3 类，数据传送指令（MOV 和 MVN）、算术逻辑运算指令（ADD、SUB 和 AND）、比较指令(CMP、TST)；

(3) 分支指令：有两种方法实现程序的跳转，一种使用分支指令直接跳转，另外一种直接 PC 寄存器赋值来实现跳转；

(4) 协处理器指令：通过 ARM 协处理器指令来实现协处理器的控制；

(5) 杂项指令：包括软中断指令 SWI、读取状态寄存器指令 MRS、写状态寄存器指令 MSR；

(6) 伪指令：ARM 伪指令不是其指令集中的指令，是编译器为了编程方便所定义的。

2.2 嵌入式操作系统概述

早期使用的 8 位处理器大多用汇编语言编写，一般都是基于应用的开发，因此也不需要区分底层代码和应用程序，维护和更新也很不方便，一般情况下如果要更改应用，则需要对代码重新编译。随着技术的发展，操作系统的运用伴随着 32 位的 ARM 处理器的出现，这样做带来了很多好处，系统地升级和维护更加方便，可扩展性和可裁减性增强，开发效率得到提高。嵌入式操作系统具有如下特点：可固化；可配置、可裁减；可修改；多版本；应用程序的开发有交叉开发工具，而嵌入式实时操作系统是嵌入式操作系统的一个重要分支。

2.2.1 嵌入式实时操作系统特点

现在主流的嵌入式操作系统都采用实时多任务操作系统 RTOS，这是和应用的复杂化相关的，随着嵌入式系统管理的外设和任务的增多，加大了操作系统管理的任务量，对中断和任务优先级的处理显得尤为重要，因此实时性成为嵌入式操作系统的重要指标。实时系统在从外界接收和处理事件时有严格的时间限制。嵌入式实时操作系统是连接硬件和应用的纽带，具有如下特点[27]：实时性得到好大的改善、可移植性增强、提供了大量的应用程序的接口、应用程序的开发更加简单。

RTOS 可以大致分为硬实时和软实时，软实时操作系统目标是任务处理足够快，没有限定必须的完成时间，硬实时操作系统则要求任务在规定的截止期完成。RTOS 不推荐用户对硬件设备和资源进行直接操作，所有的硬件设置和资源访问都要通过 RTOS 核心，用户不必清楚硬件系统的每一个细节就可以进行开发，这样就减少了开发前的学习量。

2.2.2 嵌入式实时操作系统的性能

实时操作系统对任务的运行有时间上的要求，如果是硬实时则要求更为严格，需要任务在执行时做到准时完成。很多嵌入式开发不需要操作系统支持，只需要一个 main 函数循环控制即可。在复杂度更高的应用中，RTOS 就显示出其优势，能够更好地协调任务和中断、I/O、时钟等资源的关系，使资源得到更好地利用。

一般商业化的操作系统的目标是为了保证各种资源的利用率，RTOS 的引入是为了保证任务调度的实时性和任务处理的可靠性，所以实时性是 RTOS 的主要指标，主要可以从任务调度方法、内存管理方法、任务切换和中断延迟等方面考察一个 RTOS 的实时性。

(1) 任务调度方法：RTOS 可以采用抢占式或不抢占式的调度方法，为了对突发事件做出及时响应，大部分实时性高的 RTOS 都采用抢占式的调度方法以保证实时性，及时将 CPU 控制权交给突发事件。在调度策略上，有基于优先级的调度策略和基于时间片轮转的调度策略，μC/OS-II 主要采用基于优先级的调度策略，并采用抢占式的调度方法。

(2) 内存管理方法：在有内存管理单元的操作系统中，RTOS 的内存管理方法分为实模式和保护模式，用户可以自行选择所需的模式。

(3) 任务切换：在有些地方也叫上下文切换。在发生任务切换时，RTOS 将当前状态即 CPU 寄存器的内容，保存在任务的堆栈之中。任务切换时间从保存当前任务所需要的时间一直到 RTOS 调用另一个满足条件任务到内核所需时间。

(4) 中断延迟：实时内核最重要的指标之一就是关中断的时间。在进入临界区前必须关中断，执行完代码后要开中断。中断延迟时间一般指从关中断到开始执行中断指令这段时间。中断延迟的表达式由公式 2.1 给出。

中断延迟=关中断时间+开始执行中断服务子程序第一条指令时间 （2.1）

2.3 嵌入式操作系统对多核处理器的支持

2.3.1 多核处理器概述

多核处理器在一块 CPU 板卡上合成多个处理器核心，通过总线连接各个核。事实上嵌入式双核处理器是一种比较高性价比的处理器。多核处理器比单核处理器有更多的优势，归纳起来有如下 4 点：

(1) 主频相对于单核处理器要高，能获得较高的性能。在单个时钟周期里，可以执行任务的单元数更多。但是如何高效的利用各个执行单元，使系统性能最大化是个需要解决的问题。

(2) 相对功耗低。多核集成在一个芯片中，功耗相对于 2 个单核处理器的总功耗要低。

(3) 通信延迟低。多核处理器对通信技术要求很高，多核处理器的并行技术使得通信延迟大大降低。

(4) 验证周期和设计时间比较短。处理器厂商往往会选择设计时间短、技术水平比较成熟的方法，研发性价比高的多核处理器。

2.3.2 支持多核的操作系统分类

多核处理是当前的热点技术，因此很多实时操作系统都提供了支持多核的处理方案，如

WindRiver 公司的 VxWorks6.6、QSSL 公司(QNX Software System Ltd.)的 QNX 和由 Linux 发展而来的支持多核的操作系统，这三种操作系统在实现方法上是有区别的，代表了三种典型的对多核支持的方案：

(1) WindRiver 公司的 VxWorks 采用基于同步原语的扩展方式。VxWorks 在处理器的每个核上都运行一个相同的实时操作系统，建立扩展的组件库，通过组件库中的同步原语实现各个核之间的通信。这种技术优点是扩展到多核不用做太多的修改，通过增加一个组件库就可以支持多核，缺点是增加了存储空间的需求。

(2) QNX 是一种分布式的 RTOS。它采用了微内核和分布式技术支持多核。QNX 各个模块相对独立，操作系统只提供基本的服务如任务调度和同步等。其它的服务都可以作为单独的任务和内核通信。

(3) Linux 改进后的实时操作系统。由 Linux 发展来的嵌入式 Linux 采取用一个内核来统一分配任务调度和管理各个内核的资源。

这三种方案进一步可划分为两种支持多核的操作系统模式：以 VxWorks 和 QNX 为代表的 AMP(Asymmetric Multi-Processing)模式和以嵌入式 Linux 为代表的 SMP 模式。

AMP 模式指每个处理器核上都运行一个独立的操作系统，根据运行的操作系统是否相同可以分为两种类型：同构和异构。异构的 AMP 开发比较复杂，因为需要为不同类型的操作系统建立统一的通讯方式；SMP 模式指用一个操作系统管理多个处理器核，这个操作系统负责保证用户互斥地使用各种硬件资源，协调各个处理器核的工作通讯。AMP 和 SMP 模式的结构图如图 2.2 所示：

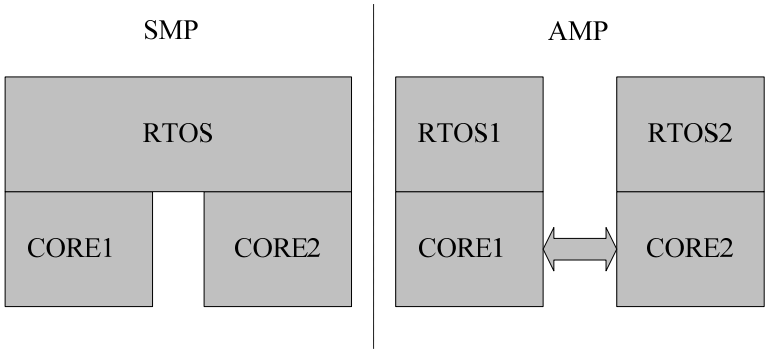

图 2.2 SMP 和 AMP 的比较

2.3.3 对称多处理器 SMP 概述

SMP 结构是多核的一种具体的实现方式。它的 N 个处理器核心完全对称地集成在系统总线上，共享主存和 I/O 设备。相比于其它异构的处理器架构，SMP 具有简单实用的特点，因此在嵌入式系统中很多并行技术的研究也是围绕着这种结构，SMP 体系结构如图 2.3 所示[28]。

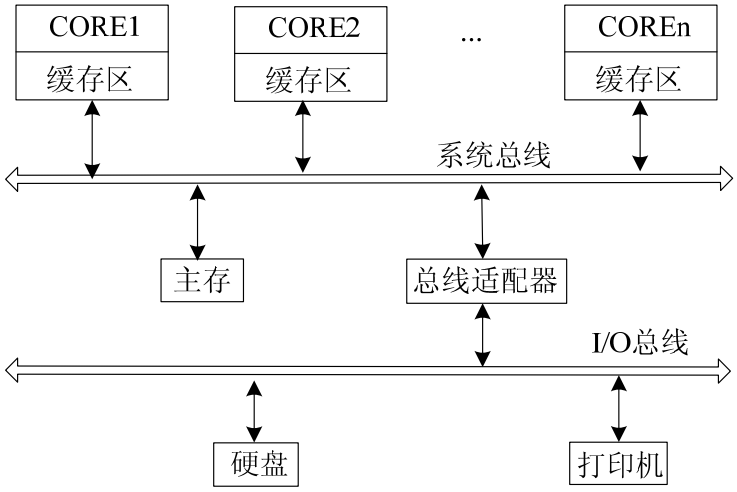

图 2.3 SMP 系统的体系结构图

具有共享主存的 SMP 系统的特点如下：

(1) 结构上对称。各个处理器核结构完全对称，每个内核都包括处理器和高速缓存，便于操作系统统一管理。各个处理器均可以访问到存储器和 I/O 设备。

(2) 可扩展性强。可以根据实际应用增减处理器，在应用需要增强处理器的并行处理能力时，可以通过增加处理器来实现。

(3) 稳定性高。由于处理器结构的对称，当某个处理器故障时系统可以继续运行不会崩溃。

(4) 处理器间通信速度快。由于每个处理器结构相同，指令集也相同。可通过简单的读写指令实现通信。

2.3.4 Linux 对 SMP 的多核处理器的支持

Linux 从 2.0 版本开始对内核进行了比较大的修改以支持对称多处理器的并行调度，一直到 2.4 版本，其中主要在初始化和启动、内核的通信、任务调度算法、中断处理和同步以及为支持并行所进行的相应数据结构的修改。Linux2.4 版本已经能很好地支持对称多处理器的要求，下面重点介绍 Linux2.4 在内核的启动、任务调度算法和中断系统处理等方面的特点。

2.3.4.1 Linux 内核的启动

在 Linux 的启动过程中[29]，必须先将应用核 AP（Application Processor）禁用，因为 BIOS 是不支持并行处理的，需要首先启动一个启动核 BP（Bootstrap Processor），在操作系统启动过程的前期只有 BP 在执行指令。 Linux 启动过程从系统加电开始，然后引导程序将内核加载到内存，初始化内存。接着调用 start_kernel，相当于应用程序中的 main()函数。start_kernel 进行一系列初始化，然后将执行 smp_init()来启动各个 AP，这是启动过程的关键步骤。最后在各个

处理器都完成自身初始化后将调度空闲任务 cpu_idle()。cpu_idle 被定义为 0 号任务，在 SMP 系统中，有几个内核，就会有几个 cpu_idle。系统的启动过程如图 2.4 所示。

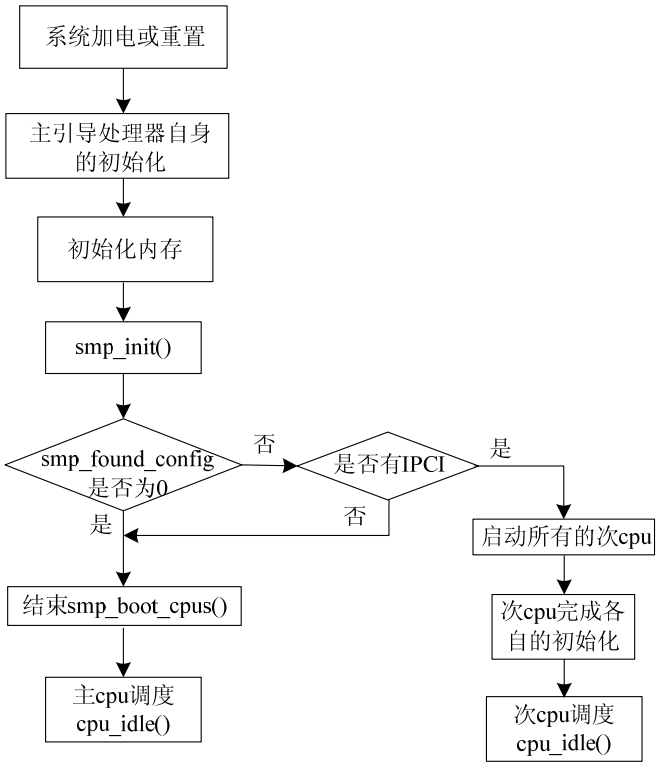

图 2.4 Linux2.4 的启动过程

Linux SMP 启动的主要流程如下：

(1) 屏蔽 AP 核，建立系统配置表格，完成 BIOS 的初始化。

(2) 通过引导程序将 Linux 内核加载到内存中。

(3) 调用 start_kernel()函数，这个函数相当于 main()函数。

(4) start_kernel()将调用 smp_init()完成 AP 的初始化，调用 rest_init()创建 1 号任务，自身执行 cpu_idle()。

(5) 1 号任务完成剩余的初始化工作。

AP 启动完成后，跳到空闲任务中，和 BP 一起等待任务调度的开始，至此 Linux 内核的启动工作完成。

2.3.4.2 Linux 任务调度器

Linux 用数据结构 task_struct 表示一个任务，与单处理器的任务控制块相比，需要增加一些新的属性值如正在使用的 CPU 和上次使用的 CPU 等，其数据结构如下[30]：

```
struct task_struct{
    volatile long states;     //当前状态，包括可运行、可中断、不可中断、停止等
    unsigned long rt_priority;   //实时任务优先级
    long counter;              //剩余时间片，开始运行时初值等于 priority
    unsigned long policy;   //调度策略有 SCHED_FIFO，SCHED_RR，SCHED_OTHER
    struct task_struct *next_task，*prev_task;      //任务双向链表
    int processor;              //正在使用的 CPU
    ……
    }
```

Linux 的任务调度区分实时任务和非实时任务。在 Linux 运行时，当 CPU 处于空闲状态或者任务的时间片用完时引起新的调度。优先级最高的任务最容易获得 CPU 的使用权，计算出的优先级用 prio 表示，任务的剩余时间片用 counter 表示，任务的优先级用 c 表示，rt_c 为实时任务的优先级。公式 2.2(a)计算非实时任务在前后 2 次运行的 CPU 不同时的优先级，公式 2.2(b)计算非实时任务 2 次运行的 CPU 相同时的优先级，公式 2.2(c)计算实时任务的优先级，宏 PROC_CHANGE_PENALTY 为 20。Linux 用 goodness()函数计算优先级，选择一个最适合的任务投入运行。

$$prio=\begin{cases} counter+(20-C) & (a) \\ counter+(20-C)+PROC\_CHANGE\_PENALTY & (b) \\ 100+rt\_c & (c) \end{cases} \qquad (2.2)$$

Linux 中，函数 schedule()是任务调度的主要函数，主要工作为[31]：

(1) 从移走任务 processor 域读取 CPU 标识，存入局部变量 this_cpu。

(2) 初始化 sched_data 变量，指向当前处理器 schedule_data 结构。

(3) 调用 goodness()函数选取任务，对于上一次也在当前处理器的任务加上 PROC_CHANGE_PENALTY 的优先权。

(4) 如果必要，重新计算动态优先权。

(5) 设置 sched_data->last_schedule 值为当前时间。

(6) 调用 switch_to()宏执行切换。

(7) 调用 schedule_tail()，如果传入的 prev 任务仍是可运行的而且不是空闲任务，schedule_tail()调用 reschedule_idle()来选择一个合适的处理器。

在 Linux2.4 版本中，采用了时间复杂度为 O(1)的任务调度算法。系统为每个处理器提供 2 个任务队列，一个是过期的任务队列，一个是活动的任务队列，每个任务队列都有 140 个优先级。任务按时间片轮转调度，时间片结束后任务由活动的任务队列进入过期的任务队列，每过

一段时间，系统执行一次负载均衡调度，重新分配任务负载。最新的 Linux2.4 与单处理器系统的主要差别是执行任务切换后，被换下的任务有可能会换到其他 CPU 上继续运行。在计算优先权时，如果任务上次运行的 CPU 也是当前 CPU，则会适当提高优先权，这样可以更有效地利用 Cache 缓存。

2.3.4.3 Linux 资源的管理

实现任务间通信最简单的方法是使用共享数据结构，但是必须保证每个任务在处理共享数据时的互斥性，以避免竞争和数据的破坏。在单处理器系统中，只要通过给处理器关闭中断就可以实现共享资源的互斥访问，在任务对所需的资源完成读写操作后再重新允许中断即可。在 SMP 中，只是禁止本地处理器核的中断不能保证对资源的互斥访问，而只能保证运行在该处理器上任务间的互斥。不仅需要禁止本地处理器核的中断，还需要处理器核间的互斥机制。

在 Linux2.4 中，系统根据任务等待资源时间的长短确定任务进入睡眠或循环等待。如果长时间无法获得所需要的资源就让任务睡眠，否则就进入循环等待。自旋锁就是这种短期互斥锁，它在短时间内反复申请资源直到申请到为止。要申请自旋锁需要调用加锁函数 spin_lock()，该函数主要通过判定锁的值是否小于等于 0，如果是说明加锁成功。否则如果锁的值大于 0 说明锁已经被释放，则跳转到返回处。解锁过程比较简单，通过调用 spin_unlock()使锁值为 1 即可。通过自旋锁的使用，Linux 很好地解决了资源管理上的问题，这种方法值得在修改 μC/OS-II 借鉴。

2.4 本章小结

本章主要介绍了嵌入式系统的组成，并重点介绍了嵌入式操作系统及其性能指标。接着分析了嵌入式操作系统对多核处理器的支持，尤其是 SMP 结构的多核处理器的特点，重点分析了支持 SMP 结构的 Linux 微内核，从 Linux 微内核的启动、并行任务调度算法设计、共享资源的互斥访问等方面分析了 Linux2.4 对支持并行结构所作的扩展工作，为 μC/OS-II 内核的改进提供了指导和依据。

第三章 嵌入式实时操作系统 μC/OS-II 的分析和改进

3.1 嵌入式 μC/OS-II 内核分析

μC/OS-II 是一款应用广泛的嵌入式实时操作系统，其内核的核心是实现任务管理。μC/OS-II 大部分源码都是 C 语言编写，可以方便地移植到各种嵌入式微处理器上。μC/OS-II 内核中仅包括任务调度、任务管理、时间管理、内存管理等基本功能。其为用户提供良好的接口，应用程序的开发可以方便地建立于其上。

3.1.1 μC/OS-II 内核结构分析

μC/OS-II 的文件结构包括 μC/OS-II 配置文件、与处理器类型无关的代码、与处理器类型有关的代码以及应用程序。其中应用程序和 μC/OS-II 配置文件需要由用户在使用应用程序时提供。与处理器类型无关的代码主要包括 μC/OS-II 的启动、任务管理、时间管理、信号量管理和内存管理等，这些模块都是作为嵌入式操作系统的内核函数的形式给出供用户使用。与处理器类型有关的代码在将其移植到不同类型的处理器时要做相应更改，系统没有给出接口函数，具体内核的结构如图 3.1 所示。

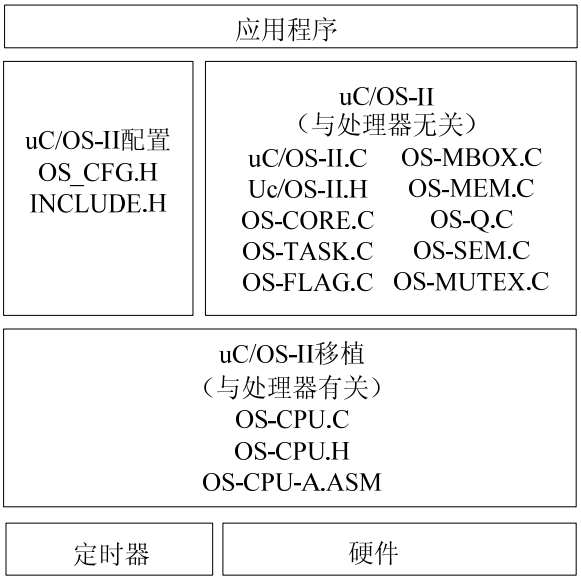

图 3.1 μC/OS-II 内核的结构图

以下具体介绍各个主要模块的功能：

μC/OS-II.C 和 μC/OS-II.H： 主函数和头文件。

INCLUDE.H 和 OS_CFG.H：系统配置文件，与应用相关。

OS_CORE.C：该模块中函数主要完成系统的初始化工作，包括系统初始化函数 OSInit()、调度器上锁函数 OSSchedLock ()、调度器解锁函数 OSSchedUnlock ()、时钟节拍函数 OSTimeTick ()等。

OS_TASK.C：该模块是、对任务状态进行操作，任务状态包括睡眠态、就绪态、运行态、等待态和中断服务态。其中包括任务的建立 OSTaskCreate()、任务的删除 OSTaskDelete()、任务的挂起 OSTaskSuspend()、任务的恢复 OSTaskResume()等。任务状态间的转换如图 3.2 所示。

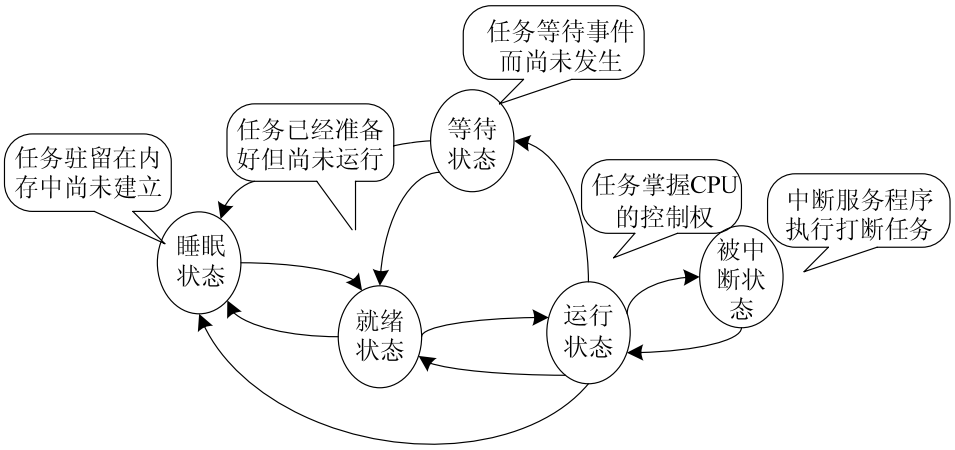

图 3.2 μC/OS-II 中的任务状态转换图

OS_TIME.C：该模块主要包括任务延时函数 OSTimeDly()、任务延时恢复函数 OSTimeDlyResume()、系统时间函数 OSTimeGet()和 OSTimeSet()。

OS_SEM.C：完成信号量的管理工作。

OS_MUTEX.C：完成互斥信号量的管理工作。

OS_FLAG.C：完成事件组标志的管理工作。

OS_MBOX.C：完成消息邮箱的管理工作。

OS_MEM.C：该模块主要完成内存管理功能。包括建立内存分区函数 OSMemcreate()、分配内存块函数 OSMemGet()和查询内存分区状态函数 OSMemQuery()。

OS_CPU_C.C 和 OS_CPU_A.ASM：是和处理器相关的代码，在移植于不同的嵌入式处理器时要做相应修改。

3.1.2 μC/OS-II 性能分析

RTOS 的使用使得应用程序的设计和扩展都变得容易，实时系统的性能主要从实时性考虑，而中断响应时间、调度延迟和上下文切换时间是衡量是实时性的重要标准。

（1）中断响应时间

对于可剥夺型内核，中断响应时间包括中断延迟时间、保存 CPU 内部寄存器的时间和内核

进入中断服务函数的执行时间。μC/OS-II 的中断处理程序允许嵌套最多到 255 层，并且是不用关中断的，中断延迟比较短，因此中断响应时间也比较短。

（2）调度延迟

调度延迟指从中断处理结束到有任务被选中开始执行这段时间。如果在中断恢复后有任务能重新被调度，那么调度延迟是有意义的。一般调度延迟的大小取决于调度算法的复杂性。在中断处理完后，不再是运行原来的任务，而是从就绪列表中找到优先级最高的任务来调度，也就是前面介绍的可剥夺型内核，由于寻找优先级最高的任务算法比较简单，因此其调度延迟比较短。

（3）上下文切换时间

上下文切换时间即任务切换时间。上下文切换时把当前任务的 CPU 寄存器中的数据保存到任务自身的堆栈中，因此上下文切换操作容易实现，一般只要数十条指令就可完成。μC/OS-II 的上下文切换时间也是比较短的。

（4）稳定性

周期性任务的基本调度能力的重要参数是处理器利用率 U，用 Ci 表示任务 Ai 的执行时间，Ti 表示任务 Ai 的周期，则 n 个周期任务的利用率由公式（3.1）给出[32]：

$$U = \sum_{i=1}^{n} U_i = \sum_{i=1}^{n} \frac{C_i}{T_i} \tag{3.1}$$

如果 U 满足下面的公式（3.2），则认为系统是稳定的。也就是说系统在满足稳定性的条件下处理器的利用率应该小于 69%。

$$U < n(2^{1/n} - 1) \xrightarrow{n \to \infty} \ln 2 \tag{3.2}$$

实际上要考虑中断和任务切换等的时间开销，设 $C^{'}$ 是中断处理时间，$T^{'}$ 是中断发生的周期，O 是调度延迟，C^{*} 表示 n 个任务的阻塞时间，则在满足公式（3.3）的条件下系统是可调度的，因为 μC/OS-II 相比于其它实时系统有较小的调度延迟和中断处理时间，因此这几个参数都很小，因而系统稳定性和实时性较好[33]。

$$\frac{C^{'}}{T^{'}} + \sum_{i=1}^{n} \frac{C_i + O}{T_i} + C^{*} < n(2^{\frac{1}{n}} - 1) \tag{3.3}$$

3.2 μC/OS-II 调度算法的研究和改进

在多核微处理器的应用环境下，操作系统的任务量肯定会有明显增加，μC/OS-II 原本设计的 64 个优先级的调度算法可能会显得具有局限性，下面主要从优先级总量和支持同优先级调度两方面考虑对 μC/OS-II 的改进，为多核情况下的扩展打好基础。在改进前，首先简要分析下 μC/OS-II 的调度机制。

μC/OS-II是一个抢先式的内核，即就绪态的高优先级任务可以剥夺正在运行的低优先级任务的CPU。这个特点使得它的实时性比较好。系统在中断处理结束时立即进行任务调度，高优先级的任务可以立即得到CPU的控制权，实时性很强。

操作系统的任务调度属于底层任务，不依赖于硬件体系结构。μC/OS-II任务调度的特点如下：具有抢占式调度功能的实时性嵌入式操作系统；支持多任务调度；一共有64个优先级（有8个是系统定义的），不支持相同优先级任务调度，即每个任务的优先级必须事先指定。如图3.3所示，通过任务优先级和任务就绪表之间的对应关系可以快速地查找到任务所在的位置，主要涉及的数据结构为：任务控制块OS_TCB、任务就绪表OSRdyGrp和OSRdyTbl[]、任务块优先级OSTCBPrioTbl[] 以及反映射表OSMapTbl[]。

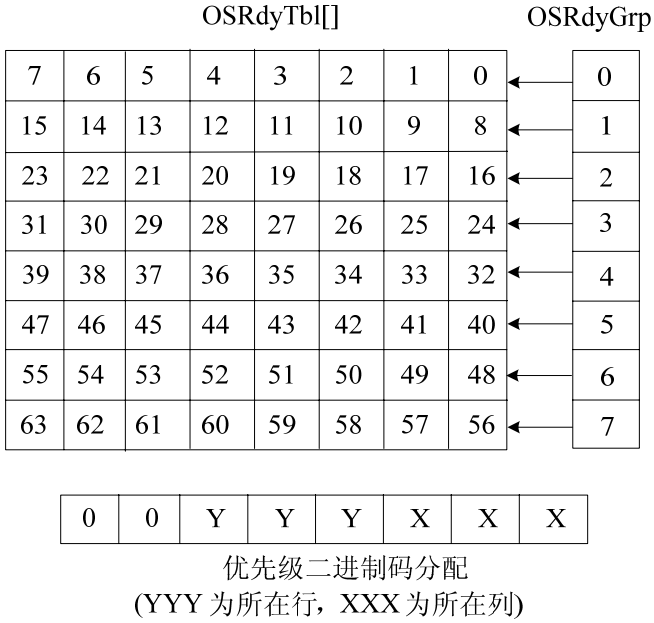

图 3.3 任务优先级和任务就绪表

3.2.1 优先级总量扩充算法的实现

如果将OSRdyGrp和OSRdyTbl[]定义为16*16的结构，可以实现优先级总量的扩充，实现μC/OS-II下256个优先级的任务调度，举例说明扩展方式。如原来一个优先级为58的任务，二进制编码为111010，如果对其扩展后编码为00111010，通过计算可得出其在任务就绪表中的位置。

因为系统定义的优先级是整型，其二进制码是8位的，而原有的优先级查找算法只使用了8位中低6位，所以可以充分的利用剩下的高2位扩展优先级总量。可以用一个8位的2进制数来表示一个优先级，这样就产生 $2^8 = 256$ 个优先级。要实现优先级总量的扩张，需要给出新的优先级查找算法。当某个优先级任务进入就绪状态后，就绪表OSRdyTbl[]的相应位置为1，当OSRdyTbl[i]中的任何一位是1时，OSRdyGrp的第i个元素置为1（ $0 \le i \le 15$ ）。

图 3.4 改进后的任务就绪表

由图3.4可以看出，改进后任务优先级低4位用于确定任务在总就绪表OSRdyTbl[]中所在位，高4位用于确定在OSRdyTbl[]数组的第几个元素。下面的代码使任务进入就绪态，prio为任务的优先级：

$$OSRdyGrp \quad |= OSMapTbl[prio >> 4]$$
$$OSRdyTbl[prio >> 4] \quad |= OSMapTbl[prio \,\&\, 0x0F]$$

其中 OSMapTbl[i]的值是一串二进制掩码，一共有 16 位，第 i+1 位为 1，其它位都是 0，如 OSMapTbl[0]=0000,0000,0000,0001，OSMapTbl[5]=0000,0000,0010,0000。

如果要使一个任务脱离就绪态，则需要做反处理。下面代码是使任务脱离就绪态，条件是OSRdyTbl[i]的所有位都是0时，OSRdyGrp的i位才是0：

$$if\,(OSGRdyTbl[prio >> 4] \,\&= \sim OSMapTbl[prio \,\&\, 0x0F] == 0)$$
$$OSRdyGrp \,\&= \sim OSMapTbl[prio >> 4]$$

在任务调度时，通过查找优先级判定表OSUnMapTbl可以找到优先级最高的任务，找出进入就绪态优先级最高的任务的代码如下：

$$y = OSUnMapTbl[OSRdyGrp];$$
$$x = OSUnMapTbl[OSRdyTbl[y]];$$
$$prio = (y << 4) + x;$$

扩展后的优先级判定表 OSUnMapTbl[]定义为一个 256*256 个数的数组，可以看成 256 行 256 列的 2 维数组，用于判定一个 16 位的二进制数非 0 位出现的最低位数。数组的元素通过 OSRdyGrp 可以得到，举例说明：若 OSRdyGrp 为 0000,1000,0001,0000，则 OSUnMapTbl[]的第 8 行第 16 列的值为 4（都是从第 0 位开始计算）。因为 μC/OS-II 优先级的高低是和数字大小成反比的，通过这种方式，可以很方便地计算出最高优先级所在位置。

3.2.2 支持同优先级算法的实现

调度的基本原则：操作系统合理分配CPU时间给就绪任务并执行该任务。任务调度应使系统保持较短的响应时间和较高的吞吐量，一个比较好的调度算法应该考虑的因素有:

(1) 保证每个任务都得到合理的CPU时间；

(2) 保持CPU的忙状态；

(3) 单位时间处理的任务数尽量多；

(4) 响应时间尽可能短。

在改变系统的调度策略前，先要搞清楚中断的作用。μC/OS-II操作系统通过中断实现任务的实时调度。在任务运行过程中，如果发生中断，将按照以下步骤进行处理：

(1) 进入中断，现场保护，即保护当前任务的执行位置，寄存器临时数据以及CPU的状态。

(2) 系统进入中断处理函数OSIntEnter()，然后执行关于中断的中断服务程序。

(3) 系统退出中断处理函数OSIntExit()，如果有更高优先级任务就绪，将引发新的调度，否则恢复现场，即恢复到原任务位置重新运行。

前人在改进μC/OS-II任务调度机制的研究上做了大量的工作[34]，其中一种很好的方法是引入时间片轮转策略。时间片轮转法的依据是任务运行的时间，通过为每个任务分配一个时间片，当某个任务的时间片完后就切换到下一任务执行。为了在μC/OS-II上实现时间片轮转法，首先要对任务控制块OS_TCB进行修改，增加同优先级任务双向循环链表以实现同优先级调度。在配置文件Os_cfg.h中增加时间片常量TIMESLICE。当某个就绪的最高优先级上有2个以上TCB时，就按给定的时间片轮转，如果有更高优先级任务进入就绪表则立即抢占CPU。

总的来说，时间片的轮转调度是一种比较成熟的多任务调度算法，μC/OS-II实现起来也比较简单。但是时间片的轮转调度算法的时间片设置是个问题，如果时间片设得太长会导致任务响应变差，如果时间片设得太短会导致任务频繁切换，降低CPU的效率以及系统实时性。下面对μC/OS-II的任务调度机制提出另一种改进方式，从而使得μC/OS-II系统很好地完成任务调度，提高系统效率。

3.2.2.1 同优先级算法的设计

根据任务调度的基本原则，对原有的任务调度方式进行改进，改进后的目的是在原有的基

于优先级抢占式调度的方式上引进同优先级调度，尽量不增加CPU用于任务切换的开销，尽可能减少内核代码的修改量。修改工作首先需要扩展原有TCB数据结构，增加TCB的双向链表结构，同优先级的任务插入双向链表中。采用双向链表结构是为了方便同优先级链表的插入和删除操作。新算法仍采用优先级抢占式调度，使得在扩充操作系统优先级的基础上实现同优先级下先来先服务调度。在μC/OS-II中优先级值越大，优先级反而越小。(算法中用到的OS_LOWEST_PRIO是在Os_cfg.h定义的，一般它的值不会超过63，系统保留了一些优先级作为扩展使用）该算法的任务创建过程如图3.5所示。

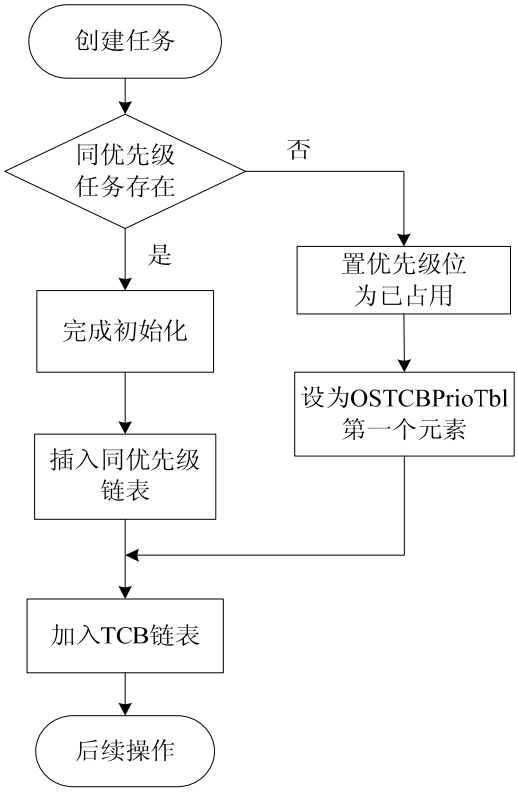

图 3.5 改进的调度策略任务创建流程

a) 任务控制块TCB的扩展

为了适应新的调度算法，首先需要对任务控制块TCB的数据结构进行修改。新的算法基本保留TCB结构体中原有的变量成员，增加TCB控制块的前向指针OSPrev和后向指针OSNext，用于两个以上的相同优先级任务的链表操作。由于系统中可能有同优先级任务存在，原有的用优先级区分任务的方法更改为用任务的ID来区分任务。新的任务控制块的数据结构定义如下：

```
typedef struct os_tcb{
        OS_STK          *OSTCBStkPtr;
```

```
    Struct os_tcb     *OSTCBNext;
    Struct os_tcb     *OSTCBPrev;
    INT16U            OSTCBPrio;
    ……
    INT16U            OSTCBId;              //任务ID号
    Struct os_tcb     *OSNext;                //同优先级任务后向指针
    Struct os_tcb     *OSPrev;                //同优先级任务前向指针
   }
```

b) 任务创建过程的改进

原有的建立任务函数OSTaskCreate()在开始时需要先关闭中断，通过OSTCBPrioTbl[prio]的值判断优先级是否被占用再创建任务，如果优先级没有被占用则完成任务堆栈的初始化和任务控制块的初始化，如果优先级已经被占用了则返回错误标志，因为一个优先级只支持一个任务。修改后的函数因为要支持同优先级任务，修改了这一判断过程，当优先级没有被占用时和过去的方法相同，当优先级被占用不返回错误标志，继续调用堆栈初始化函数和任务控制块初始化函数完成该任务的初始化，修改后的函数实现方法如下所示：

```
INT8U OSTaskCreate(){
    if(prio>OS_LOWEST_PRIO)
        return(OS_PRIO_INVALID);                    //优先级合法性判定
    OS_ENTER_CRITICAL();                            //关中断
    if(OSTCBPrioTbl[prio]==(OS_TCB*)0){          //如果该优先级没有被占用
        OSTCBPrioTbl[prio]==(OS_TCB*)1;
        OS_EXIT_CRITICAL();                         //开中断
        完成堆栈和任务控制块的初始化工作；
        任务计数器OSTaskCtr加1，调用OS_Sched()函数；
     }
    else{           //如果优先级已经被占用，则在TCB初始化时将其插入到合适的位置
        OS_EXIT_CRITICAL();
         完成堆栈和任务控制块的初始化工作；
        任务计数器OSTaskCtr加1，调用OS_Sched()函数；}
  }
```

c) TCB 的初始化过程

由于原有的内核不支持同优先级的任务，所以为了支持多个任务具有一个优先级，需要使

用新的链表结构将同优先级的任务链接起来。在任务控制块初始化时，先将其各个成员变量初始化，再将其插入到TCB链表中，该链表开始于OSTCBList，每个新任务的TCB总是插入到OSTCBList的链表头。区别于原来的算法，新的算法同时需将任务插入到同优先级链表中，该链表开始于OSTCBPrioTbl[prio]，在插入前需要判断OSTCBPrioTbl[prio]是否已经有任务，依次将每个任务插入到链表中，TCB的初始化中同优先级任务的处理过程的实现将在函数OS_TCBInit()中完成，改进后的任务控制块结构如图3.6所示。

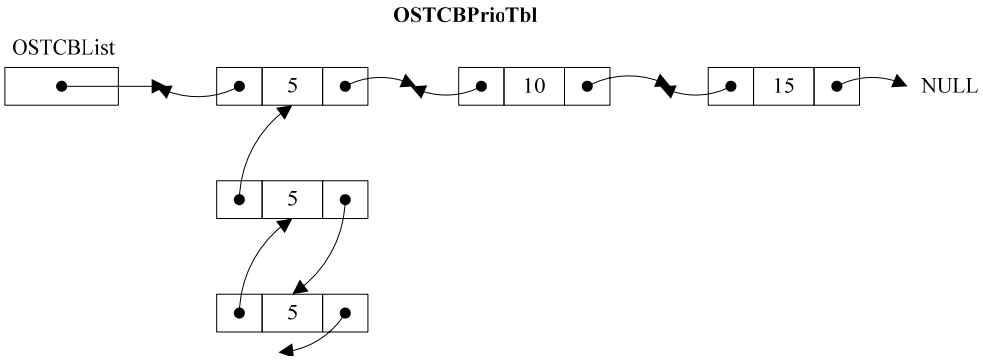

图 3.6 改进后的任务控制块结构图

```
INT8U OS_TCBInit (prio,*ptos,*pbos,id,stk_size,*pext,opt){     //传递 TCB 初始化需要的参数
    ptcb=OSTCBFreeList;                                        //从空缓冲区中得到一个任务控制块
    if(ptcb!=(OS_TCB *)0){
    传递任务控制块 ptcb 所需要的初始化参数;
    }
    if(OSTCBPrioTbl[prio] == 0) {                              //如果该优先级没有任务存在
        OSTCBPrioTbl[prio] = ptcb;
        ptcb->OSNext = ptcb;
        ptcb->OSPrev = ptcb;
     }
    else {                                                     //如果该优先级有任务存在则将其插入链表
中
        ptcb->OSnext = OSTCBPrioTbl[prio]->OSNext;
        OSTCBPrioTbl[prio]->OSNext = ptcb;
        ptcb->OSPrev = OSTCBPrioTbl[prio];
        (ptcb->OSNext )->OSPrev=ptcb;
    }
```

```
ptcb->OSTCBNext = OSTCBList；//将 TCB 链接到 OSTCBList 中
ptcb->OSTCBPrev = (OS_TCB *)0;
将该任务在就绪表中的状态更改为就绪态；
}
```

d) 任务调度过程的实现

在 μC/OS-II 中任务通常表示为一个无限循环的结构，通过调用 OSTimeDly()函数可以使任务休眠一段时间。任务调度器总是寻找优先级最高的任务进行调度，在任务调度时需要判断该任务的优先级下是否有两个及两个以上的任务存在，如果有两个以上任务存在，需要在任务调度后将该优先级的任务替换成下一个任务，由于该同优先级任务结构是个双向循环链表，所以总可以保证链表中的任务得到运行。在等待的任务因为延时而产生任务切换时还需要修改 OSTimeDly()和 OSTimeTick()来满足对任务延时的处理，需要做类似处理。

```
void OS_Sched(void){
    通过就绪表查找最高优先级的任务；
    if(OSTCBCur->OSnext != OSTCBCur){              //如果该优先级有两个以上任务
        OSTCBPrioTbl[Prio] = OSTCBCur->OSnext;     //替换成下一个任务；
        if(OSTCBPrioTbl[ptcb->OSTCBPrio]==0) {     //如果该优先级只剩下一个任务
            将该任务仍然放在链表的第一个位置；
         }
        else {
            将下个任务放在链表的第一个位置；
            }
        if(OSPrioHighRdy!=OSPrioCur){
            获取最高优先级任务，产生任务切换；
        }
      }
```

3.2.2.2 实验结果和分析

为了验证调度算法修改的有效性，将修改后的内核移植到 ARM7 系列的 LPC2200 单片机上。LPC2200 是一个基于实时仿真和嵌入式跟踪的 32 位 ARM7TDMI-S CPU，处理器时钟高达 60MHz，片内集成高速 Flash 存储器，128 位宽度的存储器接口和独特的加速结构使 32 位代码能够在最大时钟速率下运行。移植主要在 PC 机开发环境下完成调试和编译工作，参考了 ARM 系列芯片的移植标准，通过 PC 上的调试软件验证修改工作的正确性，并设计一个简单的测试

用例验证结果。移植工作主要包括：与 CPU 有关的 OSTaskStkInit( )函数的修改，初始化任务堆栈，主要设置堆栈内容；OS_CPU.H 与处理器相关的代码，包括开和关中断；OS_CPU_A.ASM 中 OSStartHighRdy( )和 OSIntCtxSw( )函数等。

表 3.1 任务状态的设置

任务名	任务 ID	任务优先级	任务延时
Task1	1	10	10
Task2	2	10	10
Task3	3	15	10

移植过程中主要修改与 CPU 有关部分的代码，修改好的代码和与 CPU 无关部分的代码一起放入编译器编译。作为可行性分析，建立 2 个优先级为 10 的任务 Task1 和 Task2，建立优先级为 15 的 Task3，3 个任务，每个任务都延时 10 个 ticks，每个任务运行过程中向串口输出一串字符，通过串口调试工具 SSCOM3.2 可以获取任务的运行情况。各个任务的状态如表 3.1 所示。

在串口调试工具 SSCOM 的终端上打印出任务的调用顺序，如图 3.7 所示，可以看到 3 个任务都可以正常运行，并且按照设计的顺序运行，因此对内核的修改达到了预期的目标。

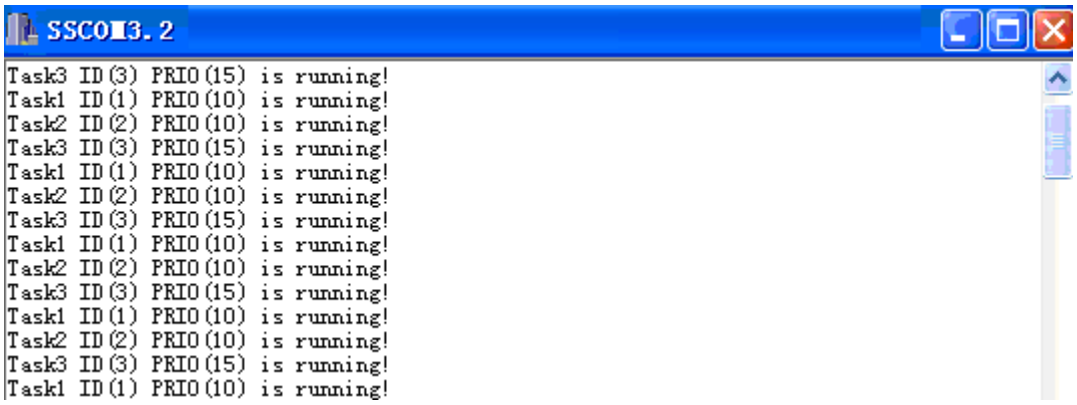

图 3.7 同优先级任务的实验结果

3.3 本章小结

本章首先介绍了嵌入式操作系统 μC/OS-II 内核的基本结构，接着介绍了实时操作系统 RTOS 的特点和性能评价标准，并分析了 μC/OS-II 的性能。接着针对在多核下的应用提出了内核 64 个优先级的总量适合轻负载的任务处理的特点，如果应用于任务量可能会较重的多处理器系统可能会显现出局限性。最后从优先级总量的扩充算法和支持同优先级调度算法两方面对这种局限性进行了改进，并通过实验验证了修改的可行性。

第四章 支持多核的 M_μCOS 微内核的设计

本章主要从软件角度设计了支持多核的嵌入式操作系统，首先根据多核处理器的特点提出了相应的设计目标并进行了 UML 建模工作，接着研究了设计过程中需要重点解决的几个问题，并提出了可行的算法设计。

4.1 M_μCOS 微内核的建模

要设计一个全新的支持多核处理器的操作系统是一个比较繁重的工作，需要大量的工作量。由于现有的 μC/OS-II 是一个比较成熟的嵌入式实时系统，可以在其内核上做一些必要的改进，使其满足多核条件下的要求。为了便于和原有的 μC/OS-II 区别，作为一个改进的后的嵌入式系统称其为 M_μCOS。

4.1.1 M_μCOS 的设计目标

嵌入式硬件环境是制约 M_μCOS 内核开发的主要因素，所以在其设计时必须考虑硬件系统的特点：嵌入式处理器的处理能力相对于 PC 来说比较弱，一般用于工业控制，所以内核设计应当优先考虑；由于嵌入式的硬件受成本的制约较多，存储器容量小，所以内核不能过于庞大。同时嵌入式操作系统也要考虑用户的需求，为用户提供良好的接口函数，方便用户申请和释放内存等操作，并且有高效的任务调度算法和高的实时性。

μC/OS-II 是一个精简的实时性很高的嵌入式操作系统，具有良好的扩展性。M_μCOS 在设计过程中保留了原来操作系统的大部分功能，这些功能主要包括：

（1）基本保留原有的数据结构。这些数据结构包括任务控制块 OS_TCB、事件控制块 ECB 等、内存控制块 OS_MEM 等，保留原有的任务就绪表结构、任务查找算法、事件处理方法和内存管理方法等，在需要时再添加多核环境下要求扩展的数据结构。

（2）支持任务优先级的抢占式调度。任务管理的基本功能保留，包括任务的创建删除、任务状态的转变、获取任务的信息。

（3）支持定时器的时钟管理和任务延时处理。定时器的时钟节拍由应用程序根据需要制定，一般可以到达毫秒级。系统可以指定当前任务延时 N 个时钟节拍，从而去执行下一个优先级最高的任务。

（4）保留原有的事件处理功能，主要包括信号量处理、消息邮箱和消息队列，但是在事件处理的实现方式上有所区别。

（5）保留了原有的内存管理功能，方便用户对内存的操作。

μC/OS-II 不支持多核处理器环境，在中断处理和任务调度等方面需要作出更改，这方面可以借鉴其它支持 SMP 架构的嵌入式操作系统的实现。M_μCOS 在保持原有系统的基本功能和实时性的前提下，需要解决下面问题：

（1）多个处理器核的启动顺序问题。以 2 个处理器核心为例，先初始化和启动一个主核，再通过启动的主核启动另外一个核，在 2 个核都启动完成的条件下再通知系统可以进入正常工作状态。

（2）多核的同步和互斥问题。由于禁止一个处理器的中断只能保证本地任务的互斥，所以在单核条件下使用的同步互斥技术不适用多处理器系统。对于共享资源，需要引入在并行处理中经常使用的自旋锁技术来实现处理器的互斥机制。

（3）多核的任务调度算法设计。由于在多核应用下任务的数量可能会明显增多，原有的 μC/OS-II 只有 64 个优先级且不支持同优先级任务的设计显得具有局限性。M_μCOS 设计了基于动态分配策略的调度算法，并且支持同优先级调度。

（4）M_μCOS 内核的移植工作。要让 M_μCOS 实时内核在 ARM 板上运行必须要做与处理器相关代码部分的修改。主要包括修改与处理器核编译器相关的代码，这些内容在 include.c 文件中；修改堆栈初始化函数 OSTaskStkInit()等；编写 OS_CPU_A.ASM 中的关于时钟中断和任务切换的汇编语言函数等。

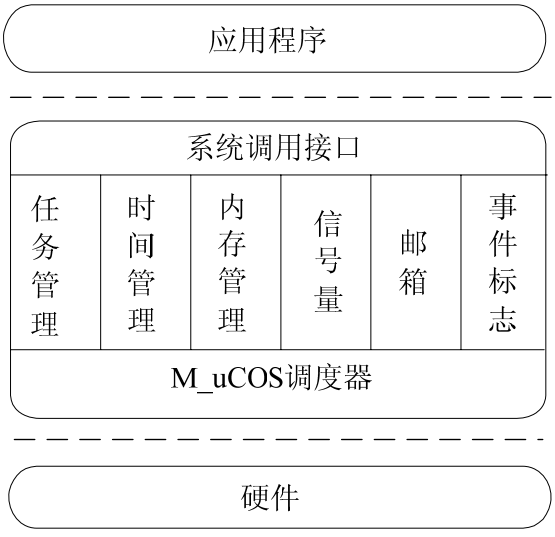

图 4.1 M_uCOS 系统架构图

本文设计的 M_μCOS 是一个微内核(Micro kernel)，只提供操作系统的核心功能。根据上面提出的功能需求，可以抽象出 M_μCOS 的系统架构，M_μCOS 提供了包括任务、时间、信号量、互斥信号量、事件组、消息邮箱、消息队列、内存等方面的管理，往下提供硬件层的接口，往

上提供应用接口，用户可以根据需要添加设备驱动、网络服务、文件系统等应用程序。M_μCOS 的系统架构图如图 4.1 所示。

4.1.2 M_μCOS 的 UML 建模

建模过程是对需要实现的系统用适当的规则简洁描述的过程，统一建模语言 UML 是一种通用的建模语言，它获得了学术界的广泛支持，事实上已经成为建模语言的标准。UML 图包括用例图、协作图、状态图等动态图和类图、对象图等静态图，在不同的图中的表现方式必须遵循一定的 UML 规则。修改原有的内核需要解决的的核心问题是任务调度过程，本章最后将对其进行设计。

为了更好地实现内核的主要组成部分及其之间的关系，本文采用了以任务为导向的面向对象的设计方法，所有的设计都是围绕着任务这个核心展开。区别于面向过程的设计方法，在建模过程中建立对象的目的不是为了完成某个具体的步骤，而是为了更好地描述该对象在整个解决问题的步骤中的行为，这样做有利于降低软件设计的难度。面向对象软件设计方法中的基本概念包括：

(1) 对象。对象是包括研究的任何事物，不仅能表示具体的事物，也可表示抽象的规则和计划。对象具有状态和行为，用具体的数据值表示状态，用一组操作表示行为。

(2) 类。类是具有相同或者相似属性的对象的抽象。可以用数据结构来表示类的属性，用操作名和实现该操作的方法来描述类的操作。

(3) 消息和方法。消息是对象之间进行通信的结构，发送一个消息至少要包括接收消息的对象名和发送给该对象的消息名。方法是类中操作的实现过程。

当前面向对象的技术在嵌入式领域的应用还比较少，嵌入式系统的开发基本上都是采用面向过程的方法，但是面向对象的分析和设计方法并不一定意味着要采用面向对象的编程语言。相反尽可能地采用面向对象的分析方法有助于保证开发过程的结构清晰，易于维护和修改。

本文在建模过程中采取了面向对象的设计思路，将整个内核所要完成的工作分解为围绕任务管理展开的几个部分，在重用 μC/OS-II 的基础上分别设计了各个部分的具体功能。

4.1.2.1 M_μCOS 用例图设计

根据以上 M_μCOS 内核的设计目标，可以抽象出 M_μCOS 各个主要角色的关系构成，这些角色单元包括中断管理、事件响应、任务管理和系统时钟：

中断管理：通知 CPU 有异步事件发生，CPU 保存现场状态到堆栈中，跳到专门的中断服务子程序 ISR 中。当 ISR 处理完后，程序回到优先级最高的任务开始执行。中断管理是嵌入式系统重要的实时保证机制，需要通过硬件来完成，能够保证对于外部事件的快速响应。对于抢占式的内核，在中断处理完成后不是进入中断的任务执行，而是进入就绪列表中优先级最高的

任务。

任务管理：用户通过创建任务实现一个具体的功能，主要包括数据处理、输入输出、请求系统服务等。实时系统的任务在运行过程中可能有高优先级的任务到来，这时候需要进行任务切换，可以通过时钟中断来检查是否需要任务切换。

事件管理：在 μC/OS-II 中，每个信号量、消息邮箱和消息队列都被分配为一个事件控制块 ECB。满足任务调度和中断管理所需要的信号或者消息，对于共享资源的管理通过关中断和自旋锁等实现对资源的互斥访问。

系统时钟：提供 M_μCOS 所需的时钟节拍，对于 SMP 系统，每个内核的时钟节拍必须同步，每个内核提供独立的时钟中断驱动该处理器上的任务切换和中断管理。在时钟初始化时，时钟计数器被 OSStart()初始化为 0，每过一个时钟节拍，计数器 OSTime 加 1。

各个角色关联着系统内的具体用例，用例及其关系列举如下：

(1) 中断响应和事件响应共同作用于中断级任务调度，当任务运行中有外部中断或请求资源时，产生中断级的调度。当产生中断后，系统运行就绪列表中优先级最高的任务。

(2) 就绪态的任务在所需资源和等待事件获得满足时，产生任务级的调度。如果该任务为最高优先级的任务，则系统执行该任务。

(3) 系统时钟根据用户设定的数值产生时钟周期，实现任务延时等操作。时钟节拍是特定的周期性中断，用户根据应用需求，设定不同的时钟节拍，一般为 10-200ms，时钟节拍频率不能太高，否则系统额外开销会增大。

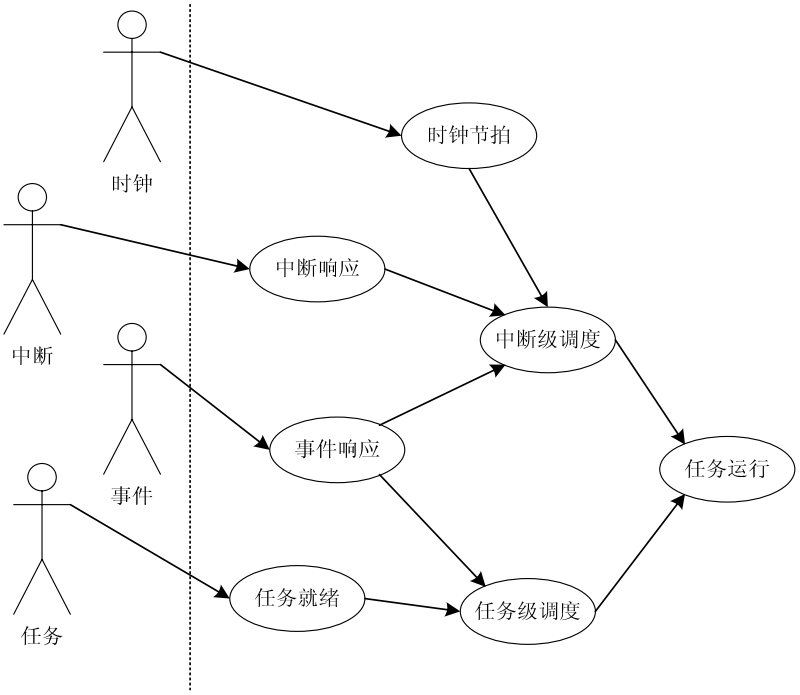

图 4.2 M_μCOS 的用例图

系统总是运行当前优先级最高的任务。任务调度器通过函数 OSSched()负责寻找优先级最高的任务运行，在运行过程中，如果有更高优先级任务到达，则通过 OS_TASK_SW()进行任务切换。中断级的调度是通过调用函数 OSIntEner()和 OSIntExit()来处理。

系统的用例图如图 4.2 所示，包括了各个角色以及角色的具体用例。其中中断级调度包括时钟中断引起的调度和任务切换引起的调度，任务级调度包括事件响应后的任务调度和调度器执行任务就绪表中的就绪任务。

4.1.2.2 M_μCOS 类图设计

UML 类图使用图形表示各个抽象类以及类之间的联系，通过对 M_μCOS 系统功能的分析，可以抽象出系统在任务处理中主要包含的类有任务类、任务调度类、中断类、事件(消息)类和定时器类，在每个类中都包含了这个类的相应的属性和相关操作以及各个类之间的数量关系。

(1) 任务包含的属性和操作：

任务 ID：任务 ID 在并行操作系统中是任务的唯一标识号；

任务优先级 Prio：任务优先级表明任务的紧迫程度，支持同优先级任务；

任务状态：任务状态包括运行、就绪、中断、睡眠等形态；

任务堆栈：任务堆栈保存任务在发生中断产生任务切换时的数据；

创建一个任务函数 OSTaskCreate()；

删除一个任务函数 OSTaskDel()；

挂起一个任务函数 OSTaskSuspend()；

恢复一个任务函数 OSTaskResume()；

(2) 任务调度包含的属性和操作：

调度策略：调度策略是基于动态分配策略的任务调度，提高系统的并行性和实时性；

任务就绪表：任务就绪表是任务调度时要查找的表结构，保存所有任务的优先级，在 M_uCOS 中的任务就绪表中；

任务调度函数 OSSHED()：任务调度函数是基于多核的调度算法，要考虑到 2 个核的负载均衡问题，与原系统的调度算法不同；

(3) 中断包含的属性和操作：

中断类型：包括外部中断、内部中断、处理器间中断和时钟中断，由于多核情况下的中断处理需要中断控制器的参与，与硬件平台的关系比较大，本文对中断的实现方式主要参考了嵌入式 Linux 系统。

中断嵌套数：在 M_μCOS 中，中断是可以嵌套的，中断嵌套数 OSIntNesting 表明该中断是处于第几层嵌套，最大值为 255。

中断处理函数 OSIntEnter()：当中断发生时，直接调用 OSIntEnter()或者 OSIntNesting，直接使得 OSIntNesting 加 1。

(4) 事件包含的属性和操作：

事件类型：M_uCOS 中用 OSEventType 定义了事件的基本类型，包括信号量、互斥信号量、消息邮箱和消息队列。

事件等待任务列表：OSEventTbl[]和 OSEventGrp 定义该事件所等待的任务 ID，它的结构和任务就绪表相似。

改变任务状态：M_uCOS 可以调用 OS_EventTaskRdy()函数使得任务进入就绪状态，调用 OS_EventTaskWait()使任务等待某个事件。

(5) 定时器包含的属性和操作：

时钟节拍：系统根据任务需要产生任务延时，当任务需要延时时，通过指定具体的时钟节拍数来完成操作。

任务延时函数 OSTimeDly()：通过提供时钟节拍 ticks 延时一个任务。

恢复延时的任务函数 OSTimeDlyResume()：可以不等待延时期满恢复任务。

M_uCOS 中各个类的属性和操作列表在 UML 类图中给出，图 4.2 中的数值表示这两个类关联之间的数量关系，从图中可以看到，任务和事件是 1 对 n 的关系，说明一个任务可以等待多个事件发生；中断和任务调度是 1 对 1 的关系，说明在中断发生后会触发任务调度。

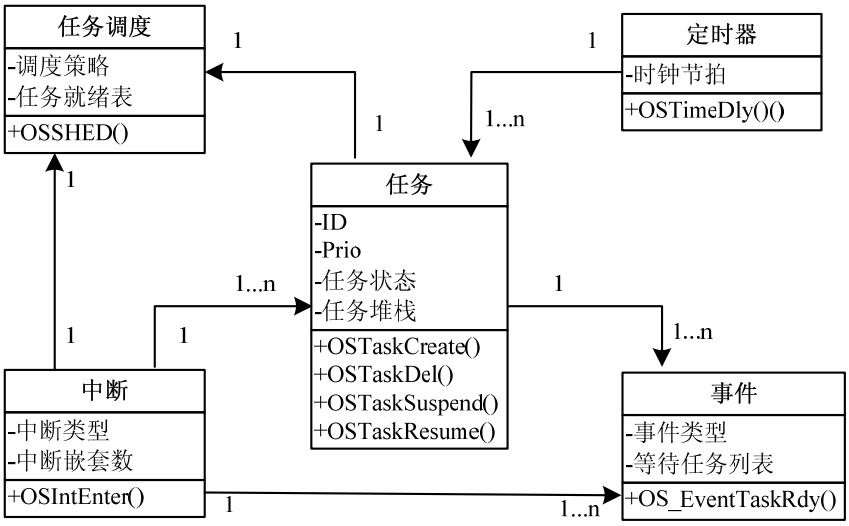

图 4.3 M_μCOS 的类图

4.1.2.3 M_μCOS 顺序图设计

顺序图是将任务的交互关系表示为一个二维图。在 M_μCOS 中，一个任务在运行过程中会发生中断或者被更高优先级任务抢占，或者会发生请求资源的情况，下面用顺序图模拟内核在

进行任务调度时的可能发生的情况，当前各个 CPU 运行的一组任务在执行过程中所遇到的情况，图中用纵坐标表示时间的变化，横坐标表示任务的迁移情况。

M_uCOS 调度过程的顺序图如图 4.4 所示。它的具体过程为：在系统正常运行后，首先有一个低优先级任务在开始执行后不久，系统提出中断服务请求，此时内核关中断，保存当前任务的 CPU 寄存器，在完成中断服务程序后，通知内核退出中断服务，首先检查是否有更高优先级的任务，如果有就跳到更高优先级的任务执行，否则返回到低优先级的任务继续执行。在任务执行过程中，可能会请求资源，如果该资源是互斥型的，则在本系统中需要为其设定自旋锁以保持其并行条件下的互斥性。

需要说明的是，其中高优先级任务是相对于当前任务而言的最高优先级的 N 个任务，如果这 N 个任务和当前任务相同，则不是高优先级任务，反之为高优先级任务。ISR 是中断服务程序，当发生中断后，CPU 将当前状态保存到寄存器中，完成中断服务后，恢复 CPU 寄存器的内容，M_uCOS 此时不是直接回到原来的任务继续执行，而需要判断当前 CPU 中运行的任务是否是具有最高优先级的 N 个任务，如果不是就会发生任务切换。这样做，保证了设计的 M_uCOS 是一个抢占式的实时嵌入式操作系统。

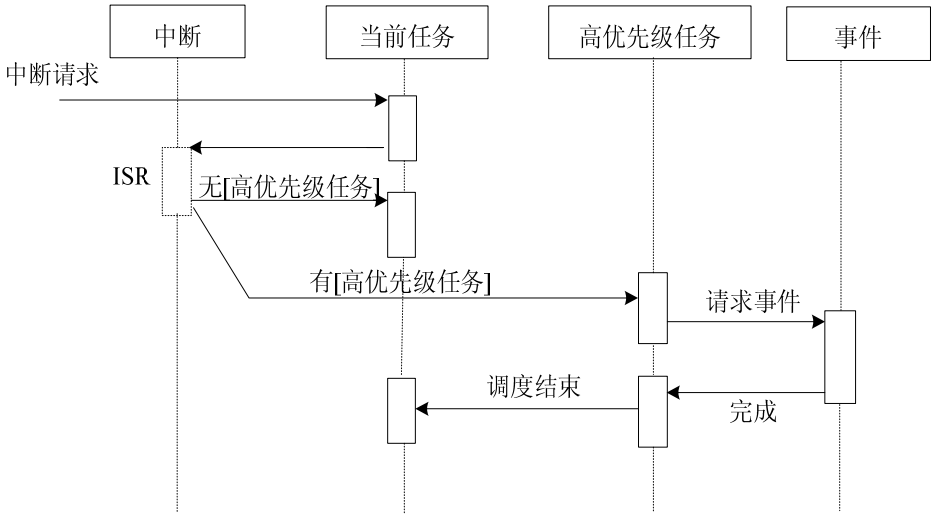

图 4.4 M_uCOS 调度过程的顺序图

4.1.2.4 M_μCOS 状态图

状态图是描述一个实体基于事件反应的动态行为，显示了该实体如何根据当前所处的状态对不同的时间做出反应。M_μCOS 的任务状态包括运行状态、就绪状态、中断服务状态、睡眠状态和等待状态。

运行状态：当前的任务被某个 CPU 调度。在单处理器中，某个瞬间只能有一个任务处于运行状态，在 SMP 系统中，可以有多个任务同时处于运行状态。当系统发生时间中断后，调度器

重新寻找当前优先级最高的 N 个任务，如果发现某个就绪状态的任务优先级高于运行状态的任务优先级时，就绪状态的任务转为运行状态，当某个运行状态任务的 CPU 被抢占后状态变为就绪状态。

就绪状态：任务建立后进入了就绪状态，准备运行。就绪状态的任务可以通过调用 OSTaskDel( )让自己或其它任务返回到睡眠状态。

睡眠状态：任务保留在存储空间中，还没有交给 M_μCOS 来管理。当调用 OSTaskCreate() 或 OSTaskCreateExt()后，告诉了系统该任务的优先级，起始地址和所需要的存储空间等信息后，任务就进入了就绪状态。

中断服务状态：正在运行的任务是可以被中断的。被中断了的任务进入中断服务状态，具体过程参考顺序图的设计过程。

等待状态：运行中的任务可以调用 OSTimeDly()将自身延时一段时间或等待某一事件发生时，任务就进入等待状态一直到延时结束火事件发生，这时 CPU 会切换到就绪状态中优先级最高的任务开始运行。当该任务等待事件结束后，系统调用 OSTimeTick()使任务进入就绪状态。

图 4.5 给出了 M_μCOS 任务状态之间的转换关系及可能要调用到的函数。如图所示，一个运行态的任务可以通过延时函数 OStimeDly()使自己延时一段时间而变为等待态，或者因为要申请某种资源变为等待态，在任务状态切换过程中可能需要申请的资源类型包括信号量、互斥信号量、消息邮箱和消息队列等。图中只列出了信号量的情况，其它情况类似处理，反之一个等待某种资源的任务因为获得了该资源而变为就绪态。

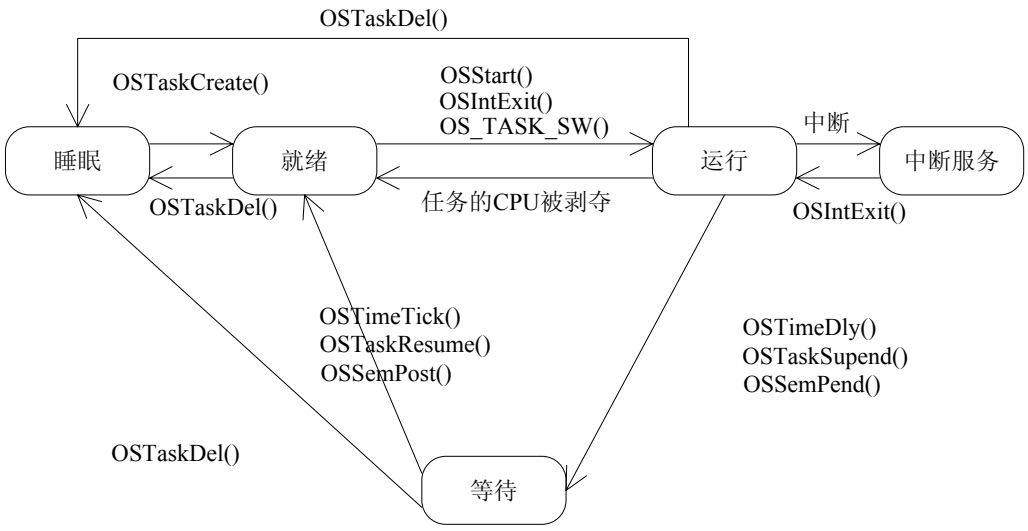

图 4.5 M_μCOS 的任务状态转换

4.2 M_μCOS 各个模块的设计

M_μCOS 的各个模块的设计过程继承了 μC/OS-II 内核的设计方法，下面简要介绍变化不大

的时间管理、事件管理、内存管理等模块的设计。

4.2.1 时间管理模块

M_μCOS 的时间管理模块提供定时中断，这个定时中断称为时钟节拍。时钟节拍在 M_μCOS 中定义为 OS_TICKS_PER_SEC，时钟节拍越高，系统负荷会越重。

M_μCOS 提供 2 个函数来处理任务延时，用户可以选择合适的任务延时方式。函数 OSTimeDly()指定任务延时的时钟节拍数，而函数 OSTimeDlyHMSM()是通过指定时、分、秒和毫秒来指定任务延时量。任务延时函数首先将当前任务从就绪表中移出，延时节拍数会被保存在当前任务的 TCB 中，每隔一个时钟节拍，OSTimeTick()将延时节拍数减 1。

M_μCOS 设置一个 32 位的计数器保存在 OSTime 中，每过一个时钟节拍计数器加 1，经过 2^{32} 个时钟节拍重新开始计数。用户可以通过 OSTimeGet()获取当前时间。在多核环境下，每个处理器都有自己的时钟计数器，系统初始化后，各个处理器的时钟计数器被初始化。

4.2.2 事件管理模块

M_μCOS 的事件管理模块用来处理任务之间的通信。通过定义事件控制块 ECB 中的 OSEventType 值可以定义事件的具体类型，M_μCOS 中的事件类型有信号量、互斥型信号量、邮箱和消息队列。任务或者中断服务子程序可以给 ECB 发信息，多个任务可以同时等待一个事件的发生，当等待事件发生后，优先级最高的任务获得该事件使用权。

M_μCOS 对等待事件的任务的处理使用了等待任务列表结构 OSEventGrp 和 OSEventTbl[]，查找方法和任务就绪表的方法相同。M_μCOS 用空余事件控制块链表来管理空余 ECB，它是一个 OSEventFreeList 为头结点的链表结构，建立一个任意类型的事件时就从其中取出一个空余的事件块，删除一个任意类型的事件时就将事件控制块放入空余事件控制块中。空余事件控制块链表如图 4.6 所示。

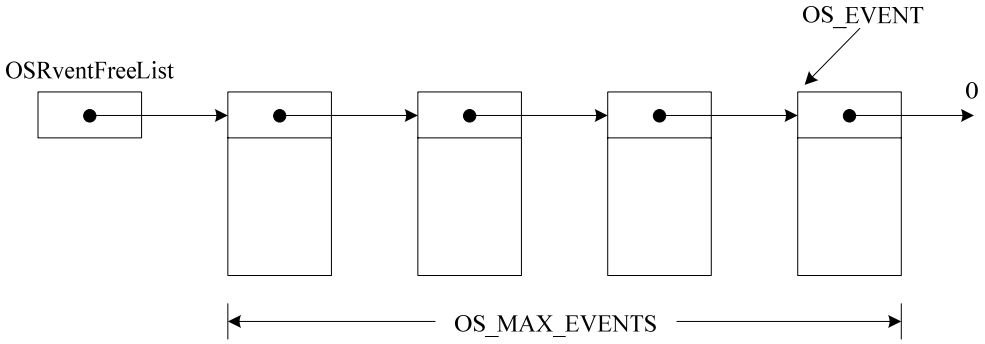

图 4.6 空余事件控制块链表

当某个任务要等待一个事件时，该事件会通过相应的 PEND 函数，使当前任务从任务的就

绪表中移出，放到相应的 ECB 等待任务表中去。M_μCOS 的事件处理函数因为事件的类型不同而不同，以信号量为例说明其事件处理过程。如果事件的 OSEventType 值设置为 OS_EVENT_TYPE_SEM，说明该事件是一个信号量。信号量的操作包括建立、删除、等待该类请求等。

OSSemCreate()：调用该函数可以创建一个信号量，首先申请到一个空余事件控制块，并赋予其初始计数值，然后让空余事件控制块指向下一个空余事件控制块。通常信号量是用来表示允许访问的相同资源的数量时，该信号量的初值为资源个数。

OSSemDel()：删除一个信号量前，需要先删除使用该信号量的的所有任务。用户可以通过 OPT 选项设置删除信号量的方式，分为强制删除和不强制删除 2 种。

OSSemPend()和 OSSemPost()：分别为让任务等待一个信号量和发出信号量给任务，当任务把优先级最高的任务从等待列表中移出后，任务进入就绪态，然后调用调度器，检查该任务是否是优先级最高的任务。

4.2.3 内存管理模块

多核系统中一般存在专门的硬件系统解决高速缓存和内存的一致性问题，系统总线会对内存操作进行监视，保证了高速缓存和内存的一致性，这种保证机制对于操作系统是透明的。M_μCOS 在内存管理时，只是简单地申请和释放一块用户所需的内存块，提供一些基本的功能。M_μCOS 使用空余内存控制块链表 OSMemFreeList 管理空闲内存块的数量。用户可以通过函数 OSMemCreate()创建一个指定大小的内存分区，使用 OsmemGet( )分配一个内存块，使用 OSMemPut()释放一个内存块。用户对内存的操作是通过内存控制块函数来实现的，内存控制块的数据结构的定义如下：

```
typedef struct {
    void    *OsmemAddr;                    //内存分区的起始地址
    void    *OSMemFreeList;                //下一个空余内存块的地址
    INT32U    OSMemBlkSize;                //内存块的大小
    INT32U    OSMemNBlks;                  //内存块的数量
    INT32U    OSMemNFree;                  //空余内存块数量
}
```

4.3 M_μCOS 中关键技术的研究

在多核系统中，要实现并行调度需要对原有内核进行一些改进，其中比较关键的有：如何保证多个处理器核心的顺利启动，使其同步开始任务调度；多核条件对资源的互斥访问不仅需要禁止本处理器上核的中断，还要处理器核间的互斥机制来保证；在多核条件下如何进行任务

调度，保证系统能正常运行的条件下任务在处理器间的切换次数较少。这些问题的解决部分参考了 Linux 系统中的方法。由于 SMP 结构的多核处理器不用固定微内核个数，在处理以下这些问题时，都采用了双内核的做为前提条件。本节主要简单研究前两个问题，第三个问题下一节将展开说明。

4.3.1 多核的启动顺序

SMP 结构的内核的启动过程与传统的单处理器核不同，单处理器只要上电后完成全局量和内存等的初始化操作后就可以执行操作系统的初始化工作，而在 SMP 中由于有两个核的存在，不能实现两个核的同时启动，必须考虑内核的启动顺序问题。

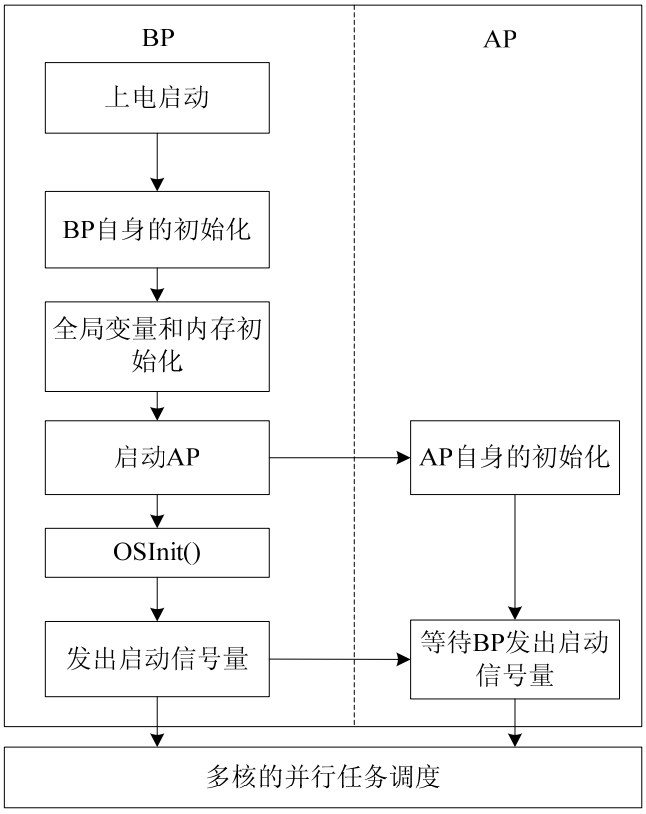

图 4.7 多核启动的流程

启动过程的设计与具体的处理器相关，这里简单介绍其思路。微内核的启动过程主要参考了 Linux2.4 系统对 SMP 的支持，在多处理器核心启动时有主次之分，首先需要启动主核 BP。以两个微内核的 SMP 为例，当系统上电后，首先启动主核 BP，另一个内核 AP 则处于未上电状态。BP 首先完成自身的初始化工作，同时完成其它全局性变量和内存的初始化，然后跳到 AP 上完成其初始化工作。对于每个核来说，自身的初始化是必须进行的，全局量的初始化主要由主核 BP 完成。BP 在完成自身初始化工作后还要调用 OSInit()完成操作系统的初始化设置，

接着调用 OSTaskCreate()或 OSTaskCreateExt()至少创建一个任务，最后调用 OSStart()开始多任务的调度，并发出启动命令。AP 一直等待到 BP 发出启动命令后才一起开始进行多任务的调度工作，保证系统任务调度开始的同步性，具体的启动过程的设计如图 4.7 所示。

在内核的启动开始时，AP 只要在接收到启动信号后完成自身的初始化工作，主要包括处理器状态字和堆栈指针的初始化，而 BP 开始启动后，需要给 AP 发送启动信号，并在完成自身的初始化工作后，完成外设和内存的初始化工作。BP 和 AP 启动完成后，都会跳到空闲任务 OS_TaskIdle()中，该任务只是不停地执行计数器加 1 操作用于统计 CPU 利用率。空闲任务永远是最低优先级的，并且不会被删除。如果有新的任务到达，此时 M_μCOS 根据设计的调度算法执行新任务。

为了实现多核处理器各个核的正常启动，需要修改内核中关于初始化和任务创建部分的代码。在 OSInit()函数中完成操作系统的初始化和空闲任务的创建，设置一个启动信号量，AP 只有收到启动信号量后才开始和 BP 一起进行任务调度。

4.3.2 多核间的互斥机制

对于单处理器的操作系统，防止中断中的并发可简单采用关中断方式。μC/OS-II 是针对单处理器的操作系统，只需要使用关中断 OS_ENTER_CRITICAL()操作和开中断 OS_EXIT_CRITICAL()操作就能保证对资源的互斥访问，所以不存在多处理器间的互斥问题。μC/OS-II 提供 3 种方法来开/关中断：第一种方法是最简单的通过指令集 CLI 和 STL 完成；第二种方法是将中断状态保存到堆栈中再关中断，通过堆栈操作存取寄存器的值；第三种方法是写一个函数，关中断时将 CPU 的状态寄存器保存到局部变量中，开中断时通过另一个函数调用局部变量恢复寄存器的值，μC/OS-II 中断机制的使用方法如下：

```
{
  OS_ENTER_CRITICAL( );      /*关中断*/
  Access the resource(read/write from/to variables);      /* μC/OS-II 临界段代码，*/
  OS_EXIT_CRITICAL( );       /*开中断*/
}
```

4.3.2.1 经典自旋锁算法

在多核系统中，不仅需要保证处理器内任务的互斥访问资源，还要保证其它处理器不会竞争该资源，必须互斥访问。M_μCOS 为了保证处理器间的互斥访问，需要引进一种在多核环境下的互斥机制自旋锁（spinlock）来解决。所谓自旋，就是在申请资源时如果该资源已经被加锁，则通过循环等待一直到该资源被释放，执行空指令就是用于延时处理。通过在申请资源前加锁，释放资源后解锁，保证了对共享资源的互斥访问。

对于互斥锁，在资源被占用的条件下，资源申请者只能进入睡眠状态。但是自旋锁不会引起调用者的睡眠，如果自旋锁被别的处理器占有，调用者一直循环在那里看是否该资源的保持者释放了锁。设计自旋锁的初衷就是在短期间内进行轻量级的锁定，一个被争用的自旋锁使得请求它的线程在等待锁重新可用的期间进行自旋。自旋锁在内核中主要用来防止多个内核并发访问临界区，防止内核抢占造成的竞争。因为自旋锁在同一时刻只能被最多一个内核任务持有，所以一个时刻只有一个线程允许存在于临界区中。

自旋锁的设计很好地满足了多核环境下所需要的锁定服务。自旋锁是用在多个 CPU 系统中的锁机制，当一个 CPU 正访问自旋锁保护的临界区时，临界区将被锁上，其他需要访问此临界区的 CPU 只能忙等待，直到前面的 CPU 已访问完临界区，将临界区开锁。自旋锁上锁后让等待线程进行忙等待而不是睡眠阻塞，而信号量是让等待线程睡眠阻塞。自旋锁的忙等待浪费了处理器的时间，但时间通常很短。自旋锁用于多个 CPU 系统中，在单处理器系统中，自旋锁不起锁的作用，只是禁止或启用内核抢占。在自旋锁忙等待期间，内核抢占机制还是有效的，等待自旋锁释放的线程可能被更高优先级的线程抢占 CPU。

M_uCOS 也采用了这种锁机制保证了多核间的互斥，自旋锁的设计包括了加锁过程和解锁过程，下面简单介绍其原理。

加锁过程（Lock）：设置一个整形变量 lock 为上锁标志，当 lock 值为 0 时代表已上锁，lock 值为 1 时代表未上锁。在内核申请该资源时需要使用加锁函数 MuCOS_SPIN_LOCK()，该函数在加锁前首先判断该锁是否被占有，如果已经被占有，则进入自旋状态，即循环判断该锁是否一直被占有直到该锁被解锁；如果没有被占有，则对其加锁使 lock 值为 0。

解锁过程（Unlock）：解锁过程相对比较简单，调用解锁函数 MuCOS_SPIN_UNLOCK()，在内核使用完该锁后将 lock 值设置为 1 即可。

图 4.8 介绍了 M_μCOS 中自旋锁加锁过程的原理，在加锁操作时，需要进行关中断操作，加锁完成后再打开中断，保证了加锁过程的原子性；也可以通过一条指令的形式完成加锁过程。在自旋锁中实现自旋的部分是对 Lock 值小于等于 0 的判断。如果 Lock 值是小于等于 0 的，则处理器不停地重复检验 Lock 值一直到 Lock 值大于 0；如果 Lock 值大于 0，表示 spin lock 已经被释放，则往回跳回到开始处，重新试图取得锁。

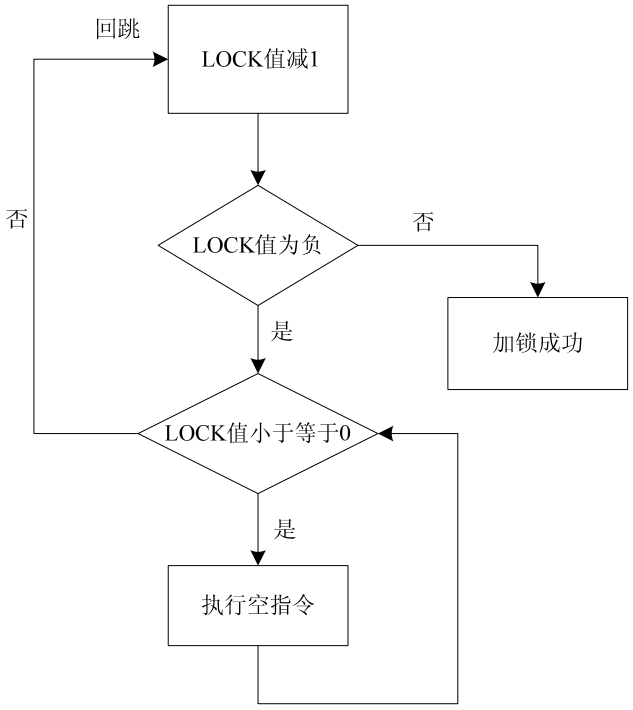

图 4.8 自旋锁加锁过程

在多核环境下往往需要将关中断和自旋锁一起使用保证临界区的互斥访问，在对临界资源访问前先要加锁保证处理器间的互斥，接着在关中断后访问临界区代码，访问完临界资源后依次开中断和解锁。自旋锁和关中断的具体使用方法如下：

```
{
    ……
    MuCOS_SPIN_LOCK(&lock);          /*加锁过程*/
    OS_ENTER_CRITICAL();             /*关中断*/
    Access the resource(read/write from/to variables);      /* M_μCOS 临界段代码，*/
    OS_EXIT_CRITICAL();              /*开中断*/
    MuCOS_SPIN_UNLOCK(&lock);        /*解锁过程*/
    ……
}
```

4.3.2.2 自旋锁算法的改进

在内核申请某个共享资源时使用自旋锁，当其它内核在使用该资源时本内核处于自旋等待状态，这种锁机制很好地保证了多个处理器之间对共享资源的互斥访问。但是仔细分析后发现这个锁的实现方式是有缺陷的，如果使用这种锁来实现共享资源的互斥访问可能造成内核的利

用率不高甚至导致死锁。

死锁是指两个或两个以上的任务在执行过程中，因为争夺资源而造成的一种互相等待的现象。产生死锁的四个必要条件为：

(1) 互斥条件：资源只能被一个任务所占有；

(2) 请求与保持条件：任务占有某个资源后因为申请其它的资源而阻塞，对已经占有的资源保持不放；

(3) 不剥夺条件：任务占有的资源在使用中不能被强制剥夺；

(4) 循环等待条件：N 个任务因为等待资源而形成循环等待状态。

自旋锁比较适合于调用者保持锁的时间相对较短的情况。在自旋锁使用过程需要解决两个问题，第一个问题是如果某个递归程序在调用时试图获得相同的锁变量，可能会造成死锁。第二个问题是某个内核的任务自旋等待时如果长时间得不到资源，会导致内核利用率不高。

第一个问题可行的解决办法是定义下面的原则：在递归程序如果想要使用自旋锁，必须保证在进入递归调用前已经释放了该自旋锁，否则会出现递归程序在占有自旋锁时调用自己或者调用相同的锁，从而产生死锁。如果一个任务已经将资源锁定，那么即使其它申请这个资源的任务不停的自旋，也无法获得资源从而进入死循环。

第二个问题可以描述为：如果内核 P_1 和 P_2 都需要某个资源 R，P_1 先占有 R，则 R 处于加锁状态，P_2 一直等待 P_1 释放 R。但是 P_1 长时间的占有 R 导致 P_2 等待时间过长，P_2 的利用率不高。这个问题的解决办法是：在 Lock 值小于等于 0 而自旋的地方不再只是执行空指令，而是设置一个计数器 i 作为判断条件，如果循环次数超过某个给定的上限 N，则跳出循环，返回一个 FALSE，否则返回 TRUE。在使用自旋锁时，需要增加判断条件，如果返回值为 TRUE，则继续进行后续对临界区资源的操作，否则发生任务切换，调用 OSTimeDly()当前任务延迟一定的时钟节拍，系统调用就绪列表中下一个优先级最高的就绪态任务。改进后的自旋锁使用方法的示意性代码如下所示：

```
{
    ……
    Ret=MuCOS_SPIN_LOCK(&lock);
    if (Ret==TRUE)
      {
       Access the resource(read/write from/to variables);      /* M_μCOS 临界段代码*/
       MuCOS_SPIN_UNLOCK(&lock);
        }
    else                              /*没有上锁成功*/
```

```
        OSTimeDly(ticks);     /*将本任务延迟，等待下次申请机会*/
        ……
    }
```

4.4 M_μCOS 调度算法的设计

在实时系统中，实时任务都联系着一个截至时间，为了保证系统正常工作，实时调度要能满足任务对截至期的要求，实现实时调度应具备下述几个条件：

(1) 提供截至时间、资源要求、优先级等必要的信息；

(2) 系统的处理能力满足实时性要求；

(3) 具有抢占式调度机制；

(4) 具有快速切换机制，包括对外部中断的快速响应和快速的任务分配能力。

从算法的复杂性角度分析，一个算法复杂性通常包括时间复杂度和空间复杂性，时间复杂性和空间复杂性的取舍要看具体的应用环境。多处理器调度算法的目标是通过增加空间复杂度的代价(如增加处理器的数量)尽量降低时间复杂度。

并行加速比是评价并行调度的重要指标，采用多处理器调度的目的就是为了获得好的并行加速比。采用多个处理器并行计算相对于单个处理器所获得的加速的倍数，Amdahl 定律定义了加速比[35] [36]。F 表示用于顺序执行部分的事件开销，P 表示处理器的个数，S 为 P 个处理器的加速比，C_1 和 C_2 为与系统性能相关的常数，它们之间的关系如（4.1）式：

$$S = \frac{1}{F + (\frac{1-F}{P}) + C_1 + C_2 P^2} \tag{4.1}$$

可见，随之处理器个数的增加，开销是二次增长的，这说明并行处理器的个数不是越多越好，存在一个瓶颈。在不考虑系统并行开销前提下，关系式可以转化为（4.2）式：

$$S = \frac{1}{F + (\frac{1-F}{P})} \tag{4.2}$$

可见顺序执行部分越小，处理器获得性能提升越大。图 4.9 为 Amdahl 定律在不考虑并行开销条件下加速比 S 与处理器个数以及顺寻执行部分 F 的关系。

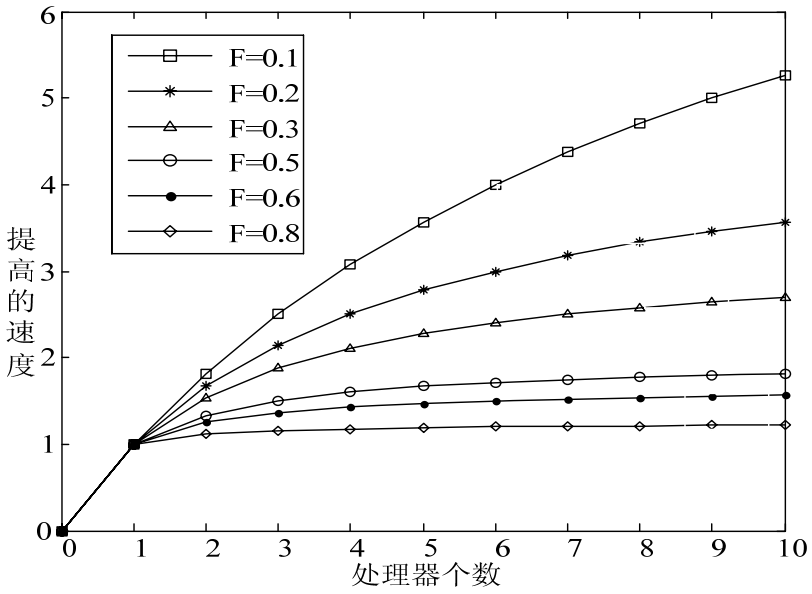

图 4.9 处理器个数对处理速度的影响

4.4.1 多核调度算法选择

多处理器系统的任务调度算法非常关键。然而多处理器调度的难点是如何确定任务分配给哪个处理器及任务分配给处理器后何时开始执行，这是一个 NP 完全问题（多项式复杂程度的非确定性问题）。如果任务为周期性任务，则相对比较好解决。如果任务为一般任务，则相对复杂。多处理器调度算法的设计要点主要有以下三点：

(1) 第一是如何把处理器分配给任务，有两种方式可供选择：静态分配策略，即一个任务永久分配给同一个处理器；动态分配策略，所有处理器公用一个就绪列表，当某个处理器空闲时，选择一个就绪任务运行。

(2) 第二是是否支持多道程序设计，即在计算机内存中同时存放几道相互独立的程序，使它们在管理程序控制之下相互穿插的运行。对于小型的嵌入式系统的应用相对比较简单，一般不需要采用多道程序设计。

(3) 第三是实际指派任务的方式。

对于多处理器的任务调度研究很多，有一些比较成熟的算法。比较常用的有时间片轮转的调度算法和单处理器扩展后的调度算法。

时间片轮转调度法：时间片轮转调度算法比较常见，如 Linux 系统就采用这种方式，它为每个任务分配一定长度的时间片，时间片用完后就切换到下个任务。轮转调度法在下述 3 种情况下才会将任务脱离运行态：任务主动放弃处理器的控制权；任务被其它任务抢占；任务自身的时间片用完。

单处理器扩展后的调度法：在单处理器中比较常见的 FCFS（先来先服务）算法、SJF（短

作业优先）算法和 EDF（最早截止期优先）算法等[37] -[39]。如果将其运用于小型的嵌入式多核操作系统是比较好的，其算法简单，修改起来比较方便，因此得到广泛应用。实验证明处理器的数目增多会导致复杂调度算法的有效性逐渐降低，因此大多数采用动态分配策略的多核系统的调度算法往往采用简单的单处理扩展后的调度法，就绪任务组成一个或者多个按优先级大小排列的队列。本文所采用的调度算法就是基于单处理器扩展后的调度算法。

4.4.2 支持同优先级的并行调度

一旦建立任务后，任务控制块 OS_TCB 就会被赋值。任务控制块是用于保存任务状态的数据结构，当任务中断恢复后，任务控制块可以保证任务的正确执行。根据 M_μCOS 内核针对于多核系统的设计要求，原有操作系统只有 64 个优先级，在并行条件下实时系统的任务量肯定会增加，如果单单通过增加优先级数不足以从根本上解决问题，M_μCOS 沿用了第三章提到的同优先级任务处理方法。

在设计多核调度算法时，需要能够正确地将各个就绪任务分配到适当的内核上运行。在分配任务时需要考虑的几个主要问题为：

(1) 任务就绪模型的选择。

比较流行的任务就绪模型有：全局就绪队列和局部就绪队列。Linux 使用是局部就绪队列，每个核有自己的任务就绪队列。而全局就绪队列是将所有的任务放到同一个就绪队列中来，每个任务都可以在任何一个内核中运行。通过分析原内核的任务调度模型，可以看出全局就绪队列更适用于 M_μCOS 的任务就绪模型，M_μCOS 通过调度器可以更容易地控制各个任务的调度，更利于实现负载均衡。

(2) 算法的时间复杂度。

实时系统的调度延时是其性能的重要指标，由于 μC/OS-II 采取的任务就绪表使得最高优先级任务选择是 O(1)级的，所以在 M_μCOS 中基于多核的任务分配算法沿用了这种结构的任务就绪表寻找最高优先级的一组任务，保证了调度算法的时间复杂性以及系统的实时性。

(3) 任务切换次数。

任务切换也叫上下文切换。当发生中断时，抢占式的实时系统会发生任务切换，寻找当前优先级最高的任务调度。在多核系统中，设计的调度算法必须保证全局任务切换次数最少，否则频繁发生任务切换必然会影响操作系统的性能。如图 4.10 所示，双核处理器各个内核当前运行的任务的优先级由大到小依次为 Task1 和 Task2，这时有优先级最高的 Task0 进入就绪队列，这时会产生任务切换。左侧发生了 2 次任务切换，是应当避免发生的任务切换算法，会降低 CPU 的性能。而如果采取了右侧的全局任务切换算法，只需要一次任务切换就可以保证任务的正确调度，极大的提高了算法的性能指标。

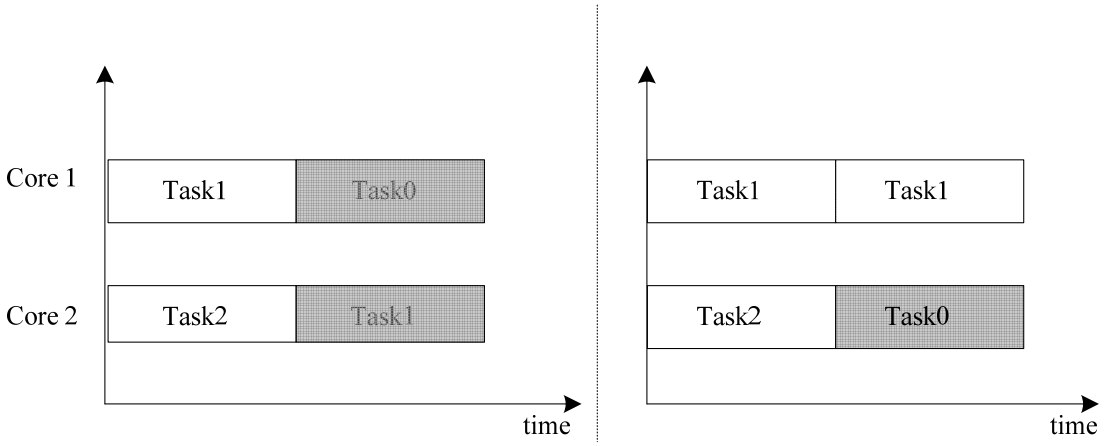

图 4.10 不同的调度算法的任务切换次数比较

综上，M_μCOS 采用的调度算法将尽量减少任务间的切换次数。M_μCOS 的任务切换可能发生在两个地方：一个是时钟中断发生时，系统需要检测每个核上运行的任务是否是优先级最高的任务，如果不是则发生任务切换；一个是任务主动延时产生的任务切换，这个处理过程需要调用任务调度器 OSSched( )函数来处理。在设计调度算法前，还需要先修改任务控制块。

4.4.2.1 任务控制块的修改

μC/OS-II 应用于单处理器系统，某个任务如果是被内核调用会将具体信息记录下来方便中断产生后的任务切换。在多处理器核的情况下情况要复杂一些，我们需要掌握该任务是否被调度以及在哪个处理器上调度，需要在为支持同优先级调度修改后的 TCB 的基础上增加新的字段来实现多核调度算法。增加一个字段 Is_Sched 标记任务是否被调度，为 1 时表示被调度中；另一个字段 Kernel_ID 标记该任务在哪个内核中被调度。修改后的 TCB 如下所示：

```
typedef struct os_tcb{
  OS_STK          *OSTCBStkPtr;
  ……
  INT16U          OSTCBId;          //任务 ID
  Struct os_tcb *OSNext;            //同优先级任务后向指针
  Struct os_tcb *OSPrev;            //同优先级任务前向指针
  INT16U          Is_Sched;         //任务是否被调度
  INT16U          Kernel_ID;        //任务在哪个内核中调度
}
```

4.4.2.2 M_μCOS 的调度模型

M_μCOS 使用任务调度器(task Scheduler)来管理任务的调度过程。当有新任务到达后加入

任务就绪表中，任务调度器通过任务就绪表可以获取合适的任务，再分配给合适的处理器调度。在单处理器环境下，任务调度器选择优先级最高的任务给处理器；在多核环境下，任务和处理器之间的选择过程是双向的，满足本文为 M_μCOS 设计的任务调度算法。当前运行的任务总是任务就绪表中优先级最高的两个任务，M_μCOS 的调度模型如图 4.11 所示。

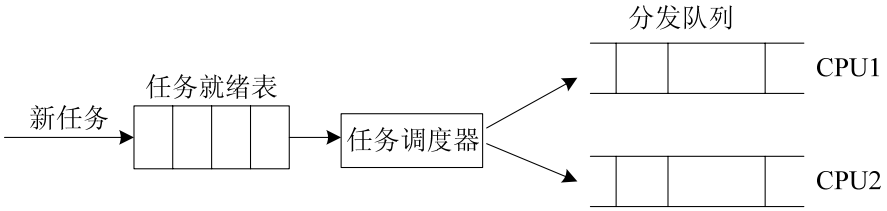

图 4.11 M_μCOS 任务调度模型

4.4.2.3 由时钟节拍中断引起的任务调度

时钟节拍源是用专门的硬件定时器实现的，在多任务启动后会开启时钟定时器。每个内核都有自己的时钟节拍定时器，当到达定时器设定的时限时就发生任务切换。内核完成时钟节拍中断后重新开始定时器的计时。

(a) 任务调度算法设计

由于有两个处理器核存在，将 OSPrioHighRdy 定义为数组 OSPrioHighRdy[i] (i=0、1)，来保存具有最高优先级的两个任务。OSPrioCur 定义为数组 OSPrioCur[i]，用来保存每个内核正在运行的任务。当最高优先级数组和正在运行的任务优先级数组不同时，就会发生任务切换，假设 B 核当前运行任务 T_1 优先级为 10，A 核当前任务 T_2 优先级为 15，任务就绪表中新进入一个优先级为 5 的任务 T_3，当前 OSPrioHighRdy[i]为[5,10]，而 OSPrioCur[i]为[10,15]，此时调度器就会产生任务切换，寻找合适的处理器调度优先级为 5 的任务。应该避免将 T_3 分配给 B 核，T_1 分配给 A 核的做法，这样会产生两次任务切换。而实际上只需要一次任务切换即在 A 核中用任务 T_3 切换任务 T_2 就可以满足要求了。M_μCOS 正是采用了这种做法进行任务调度，图 4.12 简要介绍了调度方式，这种方法可以扩展到处理器个数为 N 的情况。

这种任务调度方法的实现简单，但是却有如下优点：

(1) 保证了高优先级的任务总能获得处理器；

(2) 该算法是 O(1)级的时间复杂度，基本上不增加空间复杂度；

(3) 保证了任务切换次数较少，实现了处理器之间的负载均衡；

(4) 对原内核结构和代码的修改相对较少。

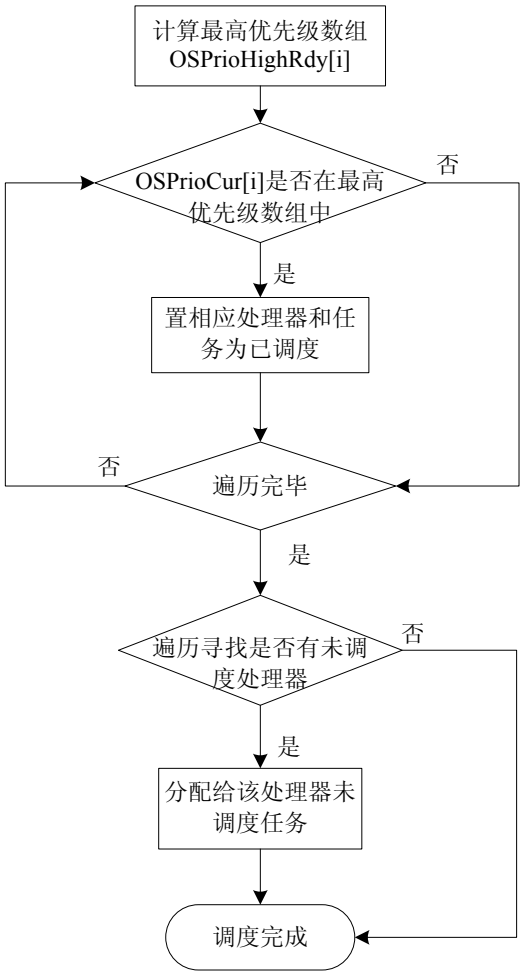

图 4.12 任务调度过程

在时钟节拍中断服务返回时 M_μCOS 会调用 OSIntExit( )函数，OSIntExit( )负责完成上述过程，其示意性代码如下所示：

```
void    OSIntExit (void){
        ……
        if (OSIntNesting > 0) {
        中断嵌套层数减 1；
        }
        if(OSPrioCur[i]!=OSPrioHighRdy[i]){
        计算出最高优先级数组 OSPrioHighRdy[i]；
        寻找到合适的处理器调度未调度任务；
        进行任务切换；
        } }
```

(b) 寻找最高优先级数组 OSPrioHighRdy[i]算法

为了不改变就绪表的状态，首先要获取就绪表的一个拷贝来方便得到最高优先级数组。在寻找优先级最高的 2 个任务时，需要判断该优先级下是否有 2 个以上的同优先级任务存在。第一种情况，如果就绪任务的最高优先级任务没有同优先级任务，则在找到最高优先级任务 a 后将置就绪表的拷贝中相应位计为 0，接着寻找次高优先级任务 b，置最高优先级数组 OSPrioHighRdy[i]为[a,b]；第二种情况，如果就绪任务的最高优先级任务 a 具有同优先级任务，则置最高优先级数组 OSPrioHighRdy[i]为[a,a]；第三种情况，如果当前除了空闲任务 c 没有其它处于就绪态的任务，则直接置最高优先级数组 OSPrioHighRdy[i]为[c,c]。同样这种算法也可以扩展到内核个数为 N 的情况。寻找最高优先级数组的算法的示意性代码如下所示。

```
{
 拷贝任务就绪表计为 OSRdyTbl_tem 和 OSRdyGrp_tem；
 y = OSUnMapTbl[OSRdyGrp];
 OSPrioHighRdy[0] = (INT8U)((y << 3) + OSUnMapTbl[OSRdyTbl[y]]); //获取最高优先级任务
 if (OSPrioHighRdy[0]==idle_id) {                                    //case 3
      OSPrioHighRdy[1]== OSPrioHighRdy[0]；
      return;
 }
 else if(OSTCBPrioTbl[OSPrioHighRdy[0]]->OSnext
                          == OSTCBPrioTbl[OSPrioHighRdy[0]]){        //case 1
    置就绪表拷贝中对应位为 0；
    在就绪表拷贝中重新寻找最高优先级任务（找到的任务是就绪表的次高优先级任务）；
    y = OSUnMapTbl[OSRdyGrp_tem];
    OSPrioHighRdy[1]= (INT8U)((y << 3) + OSUnMapTbl[OSRdyTbl_tem[y]]);
    return;
    }
  else{                                                              //case 2
      OSPrioHighRdy[i]= OSPrioHighRdy[0];
      return;
    }  }
```

4.4.2.4 任务主动延时引起的任务调度

由于 M_μCOS 处理的任务基本上为周期任务，在运行过程中周期任务可能会主动将自己延时一段时间，延时函数 OSTimeDly()会重新调用任务调度器 OSSched()引起任务切换。任务调度器重新寻找最高优先级的任务，分配合适的任务给当前处理器。通常的多核系统在任务调度器处理本次任务切换时，需要有处理器间的中断机制，在本处理器核发生任务切换时向其它处理器核发送处理器间的中断，其它处理器核在接到中断信号后转入中断服务程序执行。处理器核可在其相应状态字位设置是否需要处理器间的中断。本文设计的微内核由于增加了相应的数据结构，可以通过一个简单的循环控制完成由任务延时引起的任务调度。

在某个任务发生主动延时后，该延时任务将从就绪表中删除，任务调度器函数 OSSched()获取发生任务主动延时的内核 A 的逻辑地址，接着计算出就绪表中当前优先级最高的两个任务 T_1、T_2 的 OSPrioHighRdy[i]，看内核 B 当前运行的任务 T_3 是否在 OSPrioHighRdy[i]中。如果 $T_3 \in \{T_1、T_2\}$，不妨假设 $T_3 = T_1$，则核 A 进行任务切换运行任务 T_2；如果 $T_3 \notin \{T_1、T_2\}$，则 A、B 都需要任务切换，任务切换方式同 OSIntExit ()。任务调度器 OSSched()的处理过程如图 4.13 所示。

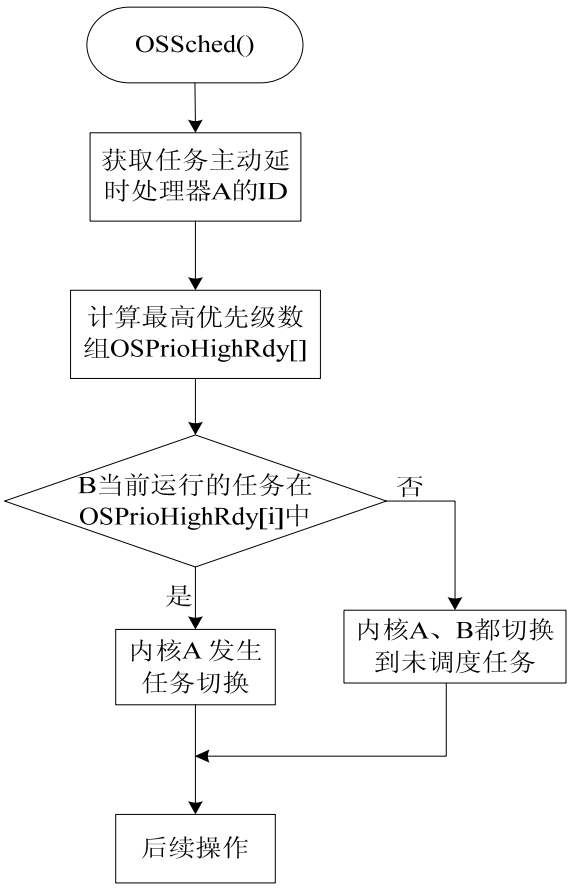

图 4.13 任务调度器的处理过程

4.4.3 调度算法的评价

本文所采用的支持同优先级的并行任务的调度算法实现简单，适用于多核环境下的任务调度，概括起来具有如下优点：

(1) 能够保证系统的实时性，高优先级的任务优先调度；

(2) 实现了多核环境下的负载均衡；

(3) 算法时间复杂度是 O(1)的，调度器的调度开销与就绪队列中的任务数无关。

(4) 在算法设计上继承了 μC/OS-II 任务调度的设计思路，保证了内核代码修改的最小化，也容易保证修改后系统的可行性。

4.5 本章小结

本章主要在参考 Linux 系统原型和第三章同优先级调度的实现算法的基础上，对 μC/OS-II 操作系统进行了研究和改进。针对多核处理器的特点，设计了微内核 M_μCOS 的用例图、类图、顺序图等实现方式，并重点考虑了 M_uCOS 在多核启动流程、并行资源管理方式和任务调度实现等方面所要做的改进工作，得出了一些有意义的实现方法。

第五章 支持多核的 M_μCOS 微内核的测试和分析

本章将介绍 M_μCOS 的硬件平台 NiosII 处理器及 NiosII 多核处理器的配置过程，介绍 M_μCOS 在 NiosII 多核处理器上移植的主要工作，并对移植后的系统进行测试，验证 M_μCOS 设计工作的有效性。

5.1 M_μCOS 的硬件平台介绍

5.1.1 NiosII 处理器简介

本文在硬件环境上使用了 NiosII 支持的多核架构[40]。NiosII 是 Altera 公司的一款软核处理器，性能超过 200DMIPS，其最大的特点是用户可以根据需要对软核进行配置。NiosII 处理器包含了一组用户可见的功能模块：如算术逻辑单元(ALU)、寄存器、控制器、JTAG 接口、指令和数据存储器接口等。NiosII 处理器的具体结构如图 5.1 所示。

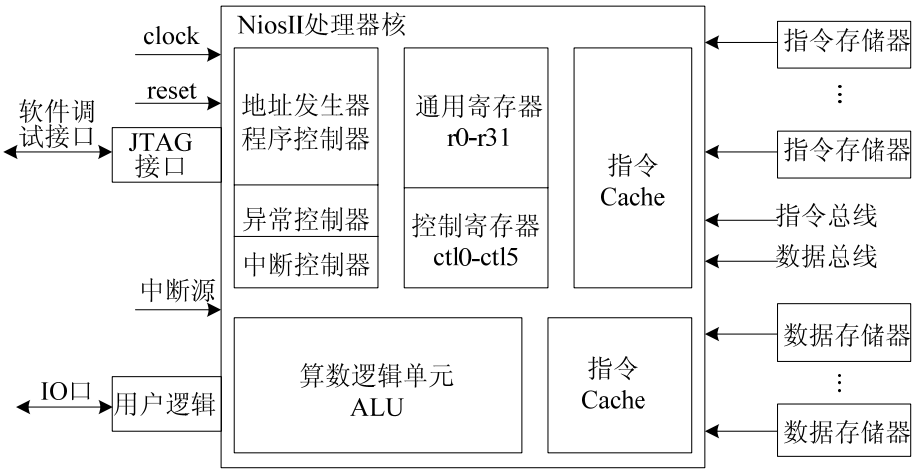

图 5.1 NiosII 处理器的结构

NiosII 处理器有三种运行模式：用户模式、超级用户模式和调试模式，调试模式拥有所有的权限，一般开发都在调试模式下进行。NiosII 有 32 个通用寄存器 r0-r31，6 个控制寄存器 ct10-ct15，控制寄存器的读写操作可以在超级用户模式下由专门的指令实现，它们的意义和用法在操作手册中详细定义。NiosII 将 IO 中断、定时器中断等交给中断控制器处理，异常事件如指令未定义等交给异常控制器处理。

5.1.2 NiosII 双核的配置过程

NiosII 支持两种多核处理器系统：自治型多处理器系统（Autonomous Multiprocessor）和共享资源型多处理器系统(Shared resources Multiprocessor)。自治型多处理器系统是指每个处理器有自己的存储器和外设，单个处理器系统之间互不相干[41]。本文所采用的 Nios II 多核处理器采用的多核架构是共享资源型的，这种资源指的是内存。该架构的缺点是不支持外部中断，因此不能共享外部设备。基于共享内存的 NiosII 多核架构如下图所示。

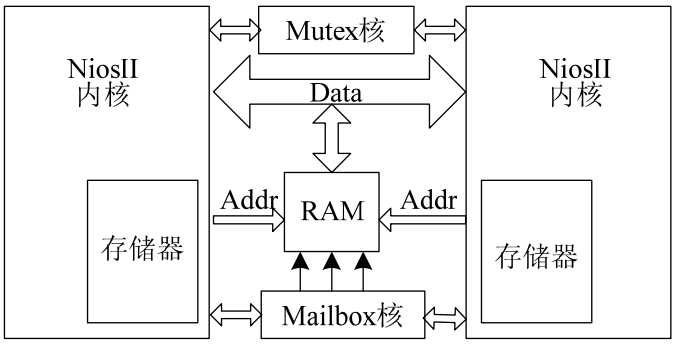

图 5.2 基于共享内存的 Nios II 多核架构

NiosII 的多核架构为用户提供了良好的技术支持，为了方便用户实现 2 个核间的同步和互斥操作，NiosII 使用了硬件 Mutex 核来协调共享资源的访问，用户可以通过 NiosII 提供的 API 函数进行操作，M_μCOS 的自旋锁就是使用其提供的 API 函数实现。NiosII 没有处理器间的中断机制来实现处理器间的通讯，NiosII 提供了 Mailbox 核来进行处理器间的通讯，用户也可以通过 NiosII 提供的 API 函数进行操作。实现同步和通信机制的 API 函数如表 5.1 所示。

表 5.1 实现同步和通信机制的 API 函数

函数名	函数功能
altera_avalon_mutex_open( )	打开互斥量
altera_avalon_mutex_lock( )	阻塞调用者，直到获得互斥量
altera_avalon_mutex_unlock( )	释放互斥量
altera_avalon_mutex _is_mine( )	判断是否拥有该互斥量
altera_avalon_mutex_first_lock( )	判断互斥量是否释放
altera_avalon_mailbox_open( )	打开 mailbox
altera_avalon_mailbox_close( )	释放资源
altera_avalon_mailbox_post( )	发送消息
altera_avalon_mailbox_pend( )	阻塞方式收取消息

NiosII 的多核架构中每个内核都有自己的内存入口地址，在内存中分配专门的空间存放代码和数据。可执行的代码放在 text 区，只读类型的数据放在 rodata 区，可读写的变量和指针放在 rwdata 中，heap 中存放的是可动态分配的内存，stack 中存放函数调用的参数和其它临时数据。具体的分配图如下所示。NiosII 双核处理器的启动流程为：一个内核完成系统的初始化工作，另一个内核等初始化工作完成后共享其数据和变量。以 2 个处理器核为例，NiosII 多核处理器的内存映像示意图如图 5.3 所示。

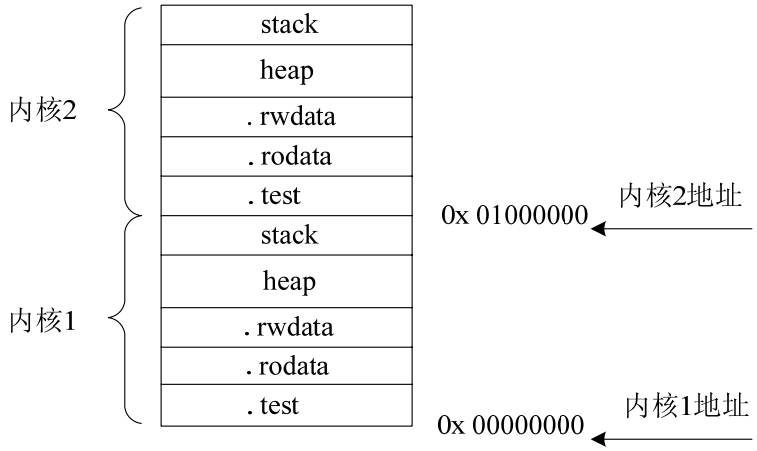

图 5.3 NiosII 双核处理器的内存映像

NiosII 软核最大的特点是用户可以根据自身的需要配置内核，这需要使用 Altera 公司提供的 Quartus II 和 SOPC Builder 工具，SOPC Builder 是一个系统级的开发工具，是 Altera 公司开发的具有革命性的产品，可以配置一个双核处理器。单核处理器的配置过程在 NiosII 的使用手册中有具体介绍，下面简单介绍多核处理器的配置过程。

(1) 从 NiosII 自带的开发包中拷贝到 neek_vic_single_91sp1_v1 文件放入指定位置，打开 Quartus II 软件载入该文件，接着打开 SOPC Builder，完成 CPU1 的配置。

(2) 增加一个新的内核。根据系统的需要选择一个 NiosII/f 型的处理器，首先要配置好 CPU2 的参数，具体的参数如表 5.2 所示。

(3) 为 CPU2 添加中断控制器、处理器时钟和消息缓存等。消息缓存由内核共享，内核可使用消息缓存通过 Mailbox 核给各个内核发送消息，Mailbox 核保证了消息缓存的互斥性。

(4) 使用 SOPC Builder 中的连接矩形连接内核间的共享资源。

(5) 添加 Mailbox 核和 Mutex 核。

(6) 建立 JTAG UART。JTAG UART 用于实现 PC 和 NiosII 串口通信的接口。

表 5.2 CPU2 的参数配置

参数	取值
NiosII Core	NiosII/f
Hardware Mutiply	None
Hardware Divide	Off
Reset Vector:Memory	ddr_sdram
Reset Vector:Offset	0x1000000
Expection Vector:Memory	ddr_sdram
Expection Vector: Offset	0x1000020
Include MMU	off
Include MPU	off

由于 NiosII 处理器拥有 Mailbox 核和 Mutex 核的硬件机制，并封装了实现任务所需要的 API 函数，使得实现多核处理器的同步和互斥变得简单。配置双核处理器的过程参考了 Altera 公司提供的技术文档《Creating Multiprocessor NiosII Systems Tutorial》，使用 QuartusII 和 SOPC Builder 对双核处理器配置完成后的界面如图 5.4 所示：

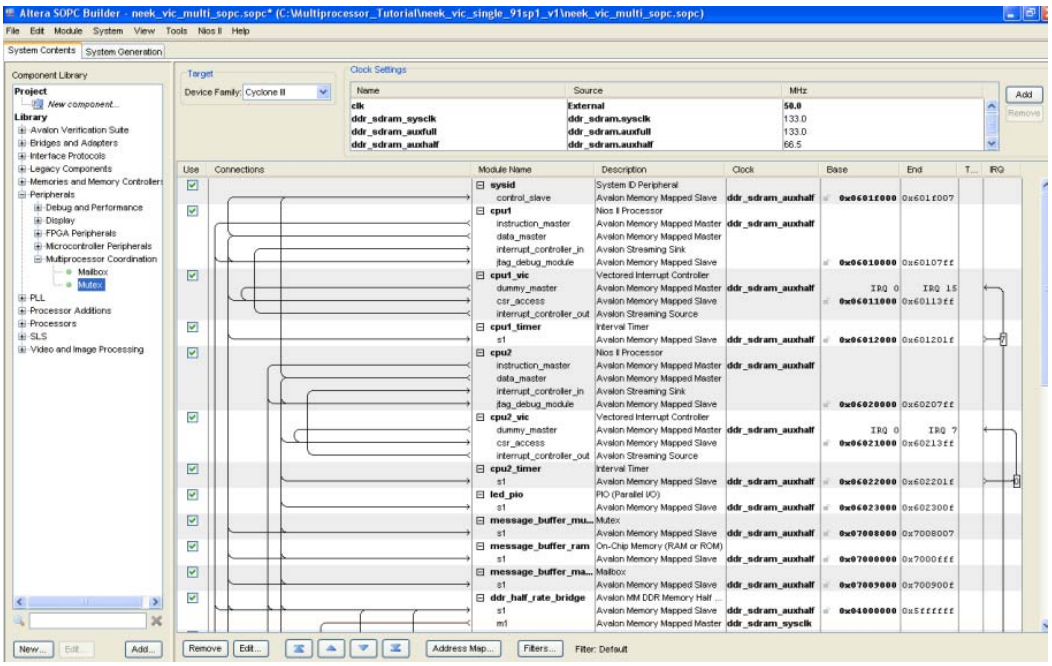

图 5.4 SOPC Builder 的配置界面

5.2 M_μCOS 的移植工作介绍

M_μCOS 继承了 μC/OS-II 的良好的可移植性，在设计上分为与处理器有关的代码、与处理

器无关的代码和与应用程序有关的代码，在移植到不同处理器平台时，只需要修改与处理器有关的代码。移植过程主要包括：与 CPU 有关的数据类型的声明、定义堆栈增长方向宏定义、进出临界区的宏定义、对临界区互斥访问的宏定义、OS_CPU_C.C 中用 C 语言写的 10 个函数、OS_CPU_A.ASM 中用汇编语言写的 4 个函数。

5.2.1 修改 OS_CPU_C.C 文件

OSTaskStkInit()：初始化任务堆栈，设置堆栈内容和保存寄存器的值，设置任务返回地址。堆栈空间由高到低依次保存各组寄存器。

OSTaskCreateHook()：空函数。

OSTimeTickHook()：可以用来设置任务任务定时器链表。

5.2.2 修改 OS_CPU_C.H 文件

完成与 NiosII 相适应的一些数据类型的定义、堆栈单位定义、定义堆栈增长方向（1 为向下）和宏定义。宏定义包括开关中断的宏定义和任务切换的宏定义。程序清单如下所示：

```
//数据类型的定义
typedef    unsigned char      BOOLEAN;
typedef    unsigned char      INT8U;
typedef    signed     char    INT8S;
typedef    unsigned short     INT16U;
typedef    signed     short   INT16S;
typedef    unsigned long      INT32U;
typedef    signed     long    INT32S;
typedef    float              FP32;
typedef    double             FP64;
typedef    unsigned long      OS_STK;
typedef    unsigned short     OS_CPU_SR;
#define    BYTE               INT8S
#define    UBYTE              INT8U
#define    WORD               INT16S
#define    UWORD              INT16U
#define    LONG               INT32S
#define    ULONG              INT32U
```

```
//关中断和开中断的宏定义
#define   OS_CRITICAL_METHOD      1
#if         OS_CRITICAL_METHOD == 1
  #define   OS_ENTER_CRITICAL()   asm("PFX 8 \n WRCTL %g0;")
  #define   OS_EXIT_CRITICAL()      asm("PFX 9 \n WRCTL %g0;")
#endif
#if         OS_CRITICAL_METHOD == 2
  #define   OS_ENTER_CRITICAL()     //保留
  #define   OS_EXIT_CRITICAL()
#endif
#if         OS_CRITICAL_METHOD == 3
  #define   OS_ENTER_CRITICAL()   cpu_sr = getITStatus()
  #define   OS_EXIT_CRITICAL()      setITStatus(cpu_sr)
#endif
//堆栈增长方向和任务切换的宏定义
#define   OS_STK_GROWTH         1
#define   OS_TASK_SW()              asm("TRAP 21");
```

在多核环境下的中断需要通过自旋锁保证处理器间的互斥，NiosII 通过 Mutex 核的硬件机制实现自旋锁操作，在头文件中加入 altera_avalon_mutex.h 即可调用系统提供的 API 函数 altera_avalon_mutex_lock()和 altera_avalon_mutex_unlock()实现多处理器的互斥。下面的代码实现了 NiosII 中的自旋锁机制，其中 mutex 是一个系统启动后被初始化的全局变量。

```
// NiosII 中自旋锁的定义
#include “altera_avalon_mutex.h”
#defined MuCOS_SPIN_UNLOCK()     altera_avalon_mutex_lock(mutex)
#defined MuCOS_SPIN_UNLOCK()     altera_avalon_mutex_unlock(mutex)
```

5.2.3 修改 OS_CPU_A.S 文件

OS_CPU_A.S 中函数都是由汇编语言实现，M_μCOS 在移植到 NiosII 时需要改写其中的 4 个函数 OSStartHighRdy()、OSCtxSw()、OSIntCtxSw()和 OSTickISR()。

(1) OSStartHighRdy()：该函数由 OSStart()函数调用，功能是运行就绪态的优先级最高的任务，该函数的示意性代码如下：

```
void OSStartHighRdy() {
```

```
    调用用户可定义的 OSTaskSwHook();
    获取任务的堆栈指针用户恢复：
    堆栈指针 ＝ OSTCBHighRdy->OSTCBStkPtr;
    OSRunnig = TRUE;
    从新任务的堆栈中恢复所有的处理器寄存器；
    从中断指令返回；
  }
```

(2) OSCtxSw()：OSCtxSw 是一个任务级任务切换函数，在任务调用中，区别于在中断程序中调用的 OSIntCtxSw()函数。

(3) OSIntCtxSw()：在中断服务子程序中，OSIntExit()函数会调用 OSIntCtxSw()，OSIntCtxSw()也称为中断级的任务切换函数。

(4) OSTickISR()：时钟中断处理函数。

5.3 M_μCOS 的测试和分析

Nios II 处理器具有完善的软件开发套件，包括编译器、集成开发环境 IDE、JTAG 调试器等。可以使用 Altera 公司开发软件中的 SOPC Builder 系统开发工具很容易地创建专用的处理器系统，并能够根据系统的需求添加 NiosII 处理器核的数量。使用 NiosII 软件开发工具能够为 NiosII 系统构建软件，自动生成适用于系统硬件的专用 C/C++运行环境。NiosII 集成开发环境 IDE 提供了许多软件模板，简化了项目设置。在完成 M_μCOS 软件的设计工作后，需要通过测试工作来验证设计工作的有效性。本节将设计测试程序和一组测试用例用于对 M_μCOS 可行性的分析。

首先需要编写测试用的任务，任务通过串口通信函数 UART_SendStr( )输出各个任务的信息。任务的格式如下所示：

```
void Task (void *pdata){
    变量的声明和初始化；
     for( ; ; ){
          UART_SendStr(str);        //输出字符串到串口
      }
     OSTimeDly(10);
 }
```

为了验证 M_uCOS 任务调度机制的有效性，设计了一组测试用例在系统中运行。由于测试用例较多，列举其中具有代表性的几个。

(1) 测试用例 1

用例设计：修改内核中的空闲任务 TaskIdle()，在空闲任务中输出其运行信息，用于显示 M_µCOS 各个内核能够正常的启动和初始化。

测试结果：NiosII 的两个内核均能正常启动并执行空闲任务，在串口调试窗口上获得如下信息”CPU1 is running!”和”CPU2 is running!”

(2) 测试用例 2

用例设计：为了验证 M_µCOS 是否支持同优先级任务，设置 2 个优先级相同的任务，看这 2 个任务是否在不同的内核上运行。

测试结果：两个同优先级任务被分配到不同的内核上调度，在串口调试窗口获得任务的信息如图 5.5 所示。

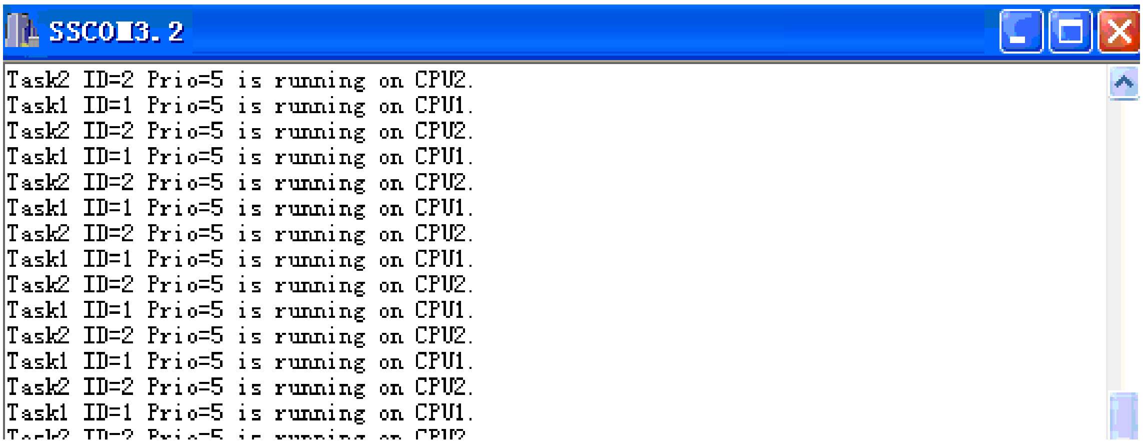

图 5.5 两个同优先级任务的调度测试

(3) 测试用例 3

用例设计：为了验证 M_µCOS 是否支持任务切换，设置一组具有不同延时的任务，看这些任务能否根据任务延时进行切换。

测试结果：测试结果表明 M_µCOS 支持本文设计的任务延时调度，当任务发生延时后会主动切换到就绪表中优先级最高的任务执行。

(4) 测试用例 4

用例设计：设置一个延时 20ticks 的任务 Task1，考察任务延时后恢复调度时选择的 CPU 是否符合本文设计的调度算法。

测试结果：测试结果表明 Task1 被调度到 CPU1 上运行，每次延时恢复后继续在 CPU1 上运行，满足 M_uCOS 的调度算法。

(5) 测试用例 5

用例设计：为了验证 M_uCOS 是否支持信号量机制，设置 2 个任务，一个任务通过

OSSemPost()发送信号量，另一个任务通过 OSSemAccept()请求信号量。互斥信号量、消息邮箱、消息队列等功能测试方法同。

测试结果：两个任务之间可以通过信号量实现通信。

上述测试用例验证了移植到 NiosII 多核处理器的 M_μCOS 符合本文的设计目标。但是由于操作系统内核结构的复杂性，对 M_μCOS 的设计和实现还有很多不完善的地方，有些功能还不支持，运行状态也不是很稳定，所以测试工作开展起来比较困难，并且只是选择最基本的功能性测试，对于内核的一些性能指标等待内核的设计完善后再进行。

5.4 本章小结

本章主要对 M_μCOS 的移植工作和测试工作进行说明，首先介绍了 M_μCOS 移植的硬件平台 NiosII 双核处理器的构造方法，接着介绍了移植 M_μCOS 操作系统所需的工作量，最后用几个简单的测试用例验证了内核设计的正确性和可行性。

第六章 总结与展望

6.1 本文的工作总结

随着嵌入式实时操作系统的广泛应用，实是操作系统应用的复杂性的不断提高，嵌入式系统向多核领域的发展已经成为一种趋势。然而目前很多嵌入式操作系统并不支持多核处理器，所以本文提出了基于 SMP 架构的多核处理器的嵌入式操作系统设计方案，围绕着多核环境展开了嵌入式操作系统内核的设计和移植工作，主要的工作内容包括：

(1) 研究了嵌入式系统的结构，分析了 Linux 系统对多核处理器的支持。本文分析了嵌入式实时操作系统的特点和发展趋势，提出嵌入式操作系统 μC/OS-II 基于多核的研究目标。分析了嵌入式系统的体系结构，从内核的启动、任务调度算法和资源管理方式等方面研究了嵌入式 Linux 系统对 SMP 的支持，对后续研究起到了借鉴作用。

(2) 通过任务总量的扩展和同优先级调度算法提高了 μC/OS-II 的任务调度能力。本文对 μC/OS-II 的内核结构和性能进行了分析，分析了 μC/OS-II 任务调度算法只能使用 64 个优先级，而且不能有同优先级任务的存在。这导致了 μC/OS-II 的任务处理能力不强，可能无法适应多核环境下的任务调度强度。本文提出了 2 种算法扩充优先级：任务总量的扩展和支持同优先级调度算法，并通过实验验证了改进的可行性。

(3) 对 μC/OS-II 进行改进，设计了一个支持多核的嵌入式操作系统 M_μCOS。根据设计目标进行 UML 建模，完成了系统用例图、类图、顺序图和状态图的设计。设计了各个模块的的实现方式，并对其中的关键技术如多核启动顺序和多核互斥机制进行了研究，重点解决了多核下任务调度算法的实现。在多核调度算法的设计上，首先选择了系统调度模型，接着修改了相应的数据结构，最后给出了多核下具体任务调度算法的实现。

(4) 通过具体的测试研究了设计工作的有效性。首先分析了硬件平台 NiosII 处理器的特点和架构，并通过工具配置了 NiosII 双核处理器。接着说明了 M_μCOS 移植到该试验平台所需的工作，并构造了一组简单测试用例测试移植后的系统，验证了设计方法的可行性。

6.2 后续工作展望

由于嵌入式操作系统实现的复杂性，需要相关的硬件和软件方面的知识比较多，本课题在研究过程中存在很多的不足，很多工作都是比较初步的，这还需要后续的学习和研究。课题进一步的研究工作如下：

(1) 当前的系统运行还不是很稳定，需要进一步完善系统的设计工作，增加对系统启动方式、异常状态等的优化，保证系统的稳定运行。

(2) 本文只是完成一个多核环境下内核的基本设计工作，还需要增加一些提高系统性能方面的研究工作，凸显出多核处理器相比于单核的优势。

(3) 本文设计的内核只提供基本的操作系统功能，是一个微内核设计，在后续工作中可以尝试添加一些模块如文件系统、网络功能和多媒体服务等，完善操作系统功能。

(4) 对系统的测试还不够全面，需要设计一些多核处理器的测试程序来全面检测系统的性能如可靠性和实时性等。

参考文献

[1] 陈国良. 并行计算. 北京：高等教育出版社, 2003：4~10.

[2] 章承科. 基于多核处理器的实时操作系统的扩展, [硕士学位论文]. 成都：电子科技大学, 2006.

[3] 刘天泉. 嵌入式系统软件设计方法研究及应用, [硕士学位论文]. 杭州：浙江大学, 2004.

[4] 毛佳. 嵌入式实时系统中关键技术的研究, [博士学位论文]. 长春：吉林大学, 2004.

[5] 杨康. 嵌入式操作系统VxWorks实时性能研究与测试, [硕士学位论文]. 长沙：国防科技大学：2009.

[6] 邢向磊, 周余, 都思丹. 基于ARMll MPCore的多核间通信机制研究. 计算机应用与软件, 2009, 5：9~10.

[7] ENSLO. P. H. Ed. Multiprocessors and Parallel Processing. New York： J. Wiley, 1974.

[8] 钟锡昌. 嵌入式操作系统在中国的发展. 中国信息导报, 2002, 5：51~54.

[9] 万柳. 嵌入式实时操作系统VxWorks内核调度机制分析. 计算机应用与软件, 2004, 6：51~52.

[10] QNX Software Systems. QNX Neutrino RTOS. http://www.qnx.com/products/neutrino_rtos/.

[11] 李善平, 刘文峰, 李程远. Linux内核2.4版源代码分析大全. 北京：机械工业出版社, 2003：1~10.

[12] 邓竹莎. 面向多处理器结构的嵌入式Linux系统研究与实现, [硕士学位论文]. 成都：电子科技大学, 2006.

[13] 诸利勇. 支持多核处理器的星载嵌入式操作系统的研究与实现, [硕士学位论文]. 长沙：国防科技大学, 2008.

[14] 邢向磊, 周余, 都思丹. 基于ARMll MPCore的多核间通信机制研究. 计算机应用与软件, 2009, 5：9~10.

[15] 周余, 都思丹. ARMll MPCore性能分析与优化研究. 南京大学学报， 2009, 1：5~6.

[16] Jean J.Labrosse. 嵌入式实时操作系统μC/OS-Ⅱ(第2版)(邵贝贝). 北京：北京航空航天大学出版社, 2003：72~282.

[17] 李瑾. μC/OS-Ⅱ操作系统的研究与应用, [硕士学位论文]. 武汉：武汉理工大学, 2007.

[18] 刘铁志. μC/OS-Ⅱ在嵌入式系统中的研究与应用, [硕士学位论文]. 成都：电子科技大学, 2005.

[19] 周立功. ARM嵌入式系统实验教程. 北京：北京航空航天大学出版社, 2006：132~135.

[20] 王永宁, 赵士杰, 齐长远. μC/OS-Ⅱ及其在ARM平台上的实现. 计算机应用与软件, 2007, 12:180~181.

[21] 崔坤, 王滨, 张文明. 基于NiosII双核系统的设计与实现. 电子技术, 2007, 6:75~78.

[22] 李正军. 基于Nios软核CPU的μC/OS-Ⅱ和LwIP移植. 遥测遥控, 2006, 2:42~47.

[23] 崔坤, 张文明, 王滨. 基于Nios II的UART与PC间的数据通信. 电子技术, 2007, 7:124~126.

[24] 吕生峰, 张光健. 基于Nios II的双核处理器的设计与实现. 中国西部科技, 2008, 31:40~42.

[25] 潘松. SOPC技术实用教程. 北京:清华大学出版社, 2005:10~30.

[26] 周立功. SOPC嵌入式系统基础教程. 北京:清华大学出版社, 2006:1~70.

[27] 曾非一. 嵌入式软件开发技术, [硕士学位论文]. 成都:电子科技大学, 2005.

[28] 夏卫民. 并行操作系统原理与技术. 北京:国防工业出版社, 2002:1~30.

[29] 李彬, 任国林. Linux内核基于对称多处理机的实现分析, 2006, 1:129~131.

[30] 洪伟, 苏晓龙, 王香婷. Linux动态调度算法的研究与实现. 大众科技, 2010, 7:28~30.

[31] 董显, 由建宏. 基于SMP的Linux系统并行性分析. 测撞技术, 2008, 27(27):69~71.

[32] 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, 6:801~814.

[33] 黄飞飞. 嵌入式实时操作系统μC/OS-Ⅱ的研究与应用, [硕士学位论文]. 南京:南京航空航天大学, 2009.

[34] 柳艳莉, 刘宏伟, 陈振华. μC/OS-II任务调度模型的分析与改进. 单片机与嵌入式系统应用, 2008, 10:20~22.

[35] 张益佳. 一种基于FPGA的MpSoC架构的设计方法和实现, [硕士学位论文]. 大连:大连理工大学, 2009.

[36] G.M.Amdahl. Validity of the Single-Processor Approach to Achieving Large-Scale Computing Capabilities. New York:ACM Press, 2004:483~485.

[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, 1:117~119.

[38] Liu C L, Layland J. Scheduling algorithms for multiprogramming in a hard real-time enviroment. J.ACM, 1973, 20(1):40~61.

[39] Martin Moln´ar. The EDF scheduler implementation in RTEMS Operating System.. Czech.

[40] 张晶. 嵌入式操作系统的设计与实现, [硕士学位论文]. 杭州:浙江大学, 2006.

[41] 张荫芾. 基于多核处理器架构的嵌入式微内核操作系统的研究与设计, [硕士学位论文]. 上海:上海交通大学, 2009.

[42] 徐光辉, 程东旭, 黄如. 基于FPGA的嵌入式开发与应用. 北京:电子工业出版社, 2006:25~55.

[43] 柳一村. 基于NIOS的SOPC系统设计以及程序引导. 电子技术, 2005, 32(6):70~72.

[44] 刘勇, 尹增山, 杨根庆. 嵌入式并行系统中基于任务优化的调度算法. 2008, 2:11~14.

[45] 贾志强. 嵌入式操作系统μCOS的移植与测试, [硕士学位论文]. 太原:太原理工大学, 2004.

[46] 宋振超. 基于多处理器嵌入式系统调度算法的研究. 电脑知识与技术, 2007, 21:777~780.

[47] RobertLove著, 陈莉君, 康华等译. Linux内核设计与实现, 机械工业出版社, 2004.

[48] 黄培镇. μCOS-Ⅱ操作系统内核研究及其工程应用, [硕士学位论文]. 成都:电子科技大学, 2006.

[49] Ramesh,Yerraballi. Real-Time operating Systems:An ongoing Review. DePt. of Computer Science and Engineering University of Texas at Arlington, 2000.

致 谢

当完成毕业论文撰写的同时，也意味着研究生阶段的即将结束。近三年的时间里我学到了很多东西，不管是专业知识还是各方面的工作经验都在一定程度上得到了提高，在这里我要深深的感谢那些指导和帮助过我的老师们、同学们。

首先要感谢我的导师徐涛老师以及同实验室的张育平老师和谷青范老师。他们孜孜不倦的钻研精神、治学严谨的工作态度以及精益求精的处事作风时时刻刻影响着我，使我整个研究生生涯都受益匪浅。在将近三年的时间里，从研究生方向的确定、专业知识的学习、实习经验的积累以及毕业论文的撰写，几位老师都给予了细心的指导，正是因为他们的帮助才避免了我在自己的学习和研究领域少走很多弯路。此外，在生活上、思想上也得到了几位老师无微不至的关心和帮助，他们丰富的人生阅历和精辟独到的见解使我对今后的发展有了更深刻的认识及更完整的规划。只言片语无法表达我此刻的心情，只能致以最真诚的谢意！

其次，同样要感谢陪我走过三年时间的 510 实验室的全体同学，特别是我的同门范海霞女士和乔乃强先生。感谢你们在生活上和学习上对我的关心和帮助，感谢你们对我的支持及鼓励。特别要感谢养育我二十多年的父母，如果没有他们精神上和物质上的支持，我无法走到今天，是他们给了我不断前进的勇气和百折不挠的决心，谢谢！

最后，衷心感谢各位老师百忙中评审我的论文，对本文提出宝贵意见。

在学期间的研究成果及发表的学术论文

攻读硕士学位期间发表(录用)论文情况

［1］ 王凯，徐涛. μC/OS-Ⅱ任务调度机制的研究. 第十七届信息论学术年会（已发表）.

基于多核的嵌入式操作系统的研究和设计

作者： 王凯
学位授予单位： 南京航空航天大学