分类号_____ 学号 M200771918

学校代码 10487 密级_____

華中科技大學

硕士学位论文

KVM 系统任务管理的设计与实现

学位申请人 严 涛

学 科 专 业：计算机系统结构

指 导 教 师：林 安 副教授

答 辩 日 期：2009.5.25

A Thesis Submitted in Partial Fulfillment of the Requirements

for the Degree of Master of Engineering

Design and Implementation of Task Management of KVM System

Candidate : Yan Tao

Major : Computer Architecture

Supervisor : Assoc. Prof. Lin An

Huazhong University of Science & Technology

Wuhan 430074, P.R.China

May, 2009

独创性声明

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

学位论文作者签名：

日期：　　年　　月　　日

学位论文版权使用授权书

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

本论文属于　保密□，　　在_____年解密后适用本授权书。

　　不保密□。

（请在以上方框内打“√”）

学位论文作者签名：　　指导教师签名：

日期：　　年　　月　　日　　日期：　　年　　月　　日

摘 要

随着现代科技的发展，通用操作系统在某一特殊领域的性能显得越来越疲乏，专门服务于某一具体应用的专用操作系统已经成为近年来新兴的研究和应用热点。主要的研究工作是设计和实现一个名称为 KVM OS 的专用操作系统中的任务管理模块。KVM OS 是一款专门为商业应用软件“KVM 切换器”提供操作系统级服务的专用操作系统。

首先结合应用软件 KVM 切换器的应用背景分析了 KVM 系统任务管理机制的需求，对任务管理机制进行了全面的研究，尤其是对如何提高 KVM 系统整体的实时性这个问题做了分析和研究。

其次给出了 KVM OS 任务管理详细的设计和实现过程。在多任务上下文切换管理方面，结合平台 ARM9 处理器的特点设计了一种处理器和堆栈之间传递数据的模型，有效地加快了上下文切换的速度。在定时器管理方面，设计了一种定时机制模型。这种模型包括由定时器任务控制块组成的数据结构和对该数据结构进行操作的算法，能够使定时器触发任务以此保证了系统的实时性。针对实时操作系统抢占内核的特点，设计出了一种基于优先级的抢占调度算法，在基于多任务的情况下能够满足时间复杂度低同时保证了调度的公平性。

最后对于 KVM OS 任务管理机制从功能和性能两个方面进行了测试和分析。实验结果表明，多任务的上下文切换平均时间和一些著名嵌入式操作系统接近，定时器定时功能具有较低的时延，任务调度能够满足调度的正确性。验证了 KVM OS 任务管理机制具备较强的实用性。

关键词：任务管理，任务调度，任务切换，定时器管理

Abstract

With the development of science and technology General-purpose Operating System becomes more and more poor in the face of some specific environment ， Special Operating System which was served specially for a s specific application becomes the focus of research and application in recent years. The main contributions of this thesis lie in the design and implementation of the task management module of a Special Operating System called “KVM OS”. KVM OS is a Special Operating System which specially provides the business software appliance “KVM-switching” with services at operating system-level.

Firstly, the thesis analyses the requirement of task management of the KVM OS on the base of the background of KVM-switching, does comprehensive research on task management, especially on the problem of improving the real-time performance of the whole KVM system.

Secondly, the thesis presents the process of the design and implementation of the task management module of KVM OS in detial. In order to handle the multitasks’ context-switching,the thesis designs a model of data communication between processor and tasks’ stack according to the feature of ARM9 processor.This accelerates the context switching effectively.In the part of timer management,the thesis designs a model of timing.This model includes a data structure which is composed of timer’s PCB and a arithmetic of instructions on how to operate the data structure.It enables multitask to be activated by timers so as to guarantee the real-time performance of the system. Considering the preemptive core of the Real-Time Operating System, the thesis designs a priority-based scheduling method which adapt to the preemptive scheduling.With low time-complexity, the algorithm ensures the equity of scheduling in the condition of multi-task.

Finally,task management of KVM OS is tested and analyzed in two aspects which are functions and performance.The results shows that the average time consumed by the context-switching is much the same as some famous Embedded Operating System. The timing function has a little delay. Task schedule assures its reliability. These show that the task management module of KVM OS is of relatively high utility.

Key words: Task management, Task scheduling, Task-switching, Timer management

目 录

1 绪论

专用操作系统Special-Purpose Operating System[1]是一类专门应用于特殊领域的计算机操作系统。随着信息科技的高速发展，专用操作系统和人类的生活联系的越来越紧密。复杂的应用环境对专用操作系统的稳定性和可靠性要求较高。因此，专用操作系统的研究成为了一个非常有意义的课题。

1.1 课题来源、目的与意义

1.1.1 课题来源

本课题来源于实验室课题组与某公司合作项目“KVM 切换器专用操作系统 KVM OS” 。

1.1.2 课题目的及意义

KVM 切换器的功能是用一组键盘、显示器和鼠标控制多台远程计算机主机。KVM 切换器的应用领域已经扩展到串口设备，如集线器、路由器、存储设备等。精密的 KVM 切换器解决方案可以让多位使用者在任务地点、任何时间访问数以千计的服务器和网络设备。帮助使用者从它们希望的任务地点安全地管理日常的运作。作为一款商用服务器管理软件的 KVM 切换器对操作系统有较高的要求，除了稳定性，实时性，可移植性等还包括价格等诸多因素。

当今国外的嵌入式操作系统已经比较完善，著名的开源嵌入式操作系统包括uC/OS-II以及LINUX，项目小组起初打算直接使用这两者中的一种。然而经过分析后认为这两个操作系统内核都有各自的缺陷，不适合作为服务于KVM切换器的操作系统。uC/OS-II主要的不足之处在于：用户必须为每个应用指定一个唯一的优先级[2]。LINUX的内核是不可抢占的，这使得LINUX不能很好地支持响应速度要求高的实时应用[3]。LINUX内核运行在嵌入式硬件平台时需要添加实时软件模块。这些模块是在内核空间运行的，代码错误可能会破坏操作系统从而影响这个系统的可靠性和稳定性，这对于实时应用是一个严重的弱点[4]。由于没有找到一款直接能够使用的嵌入

式操作系统，本课题组研发了一款服务于KVM切换器的专用操作系统KVM OS。

KVM OS整个系统划分为任务管理模块、任务通信模块、内存管理模块。针对可移植性是对嵌入式操作系统的一个重要要求[5]这一特性，将源代码的编写分为处理器关联部分和处理器无关部分。处理器关无关部分采用C语言编写，处理器关联部分采用ARM9 汇编语言编写，同时尽可能的精简汇编语言的代码量，提高KVM OS的可移植性以及处理器关联部分的运行速度。

KVM OS 是一个支持多任务的嵌入式实时操作系统。硬件平台选择了基于ARM9 处理器的嵌入式开发板S3C2410。S3C2410 是典型的 32 位RISC芯片，具有体积小、功耗低、运算速度快、片内集成度高等优点[6]。KVM OS的实现为KVM切换器应用层提供了良好的操作系统级服务，通过以应用程序接口API[7]的形式给KVM切换器应用层提供服务。

1.2 国内外概况

嵌入式操作系统因其原代码量小，高可靠性等特性，目前已广泛应用于信息业，国防，通讯行业等诸多领域[8]。目前的嵌入式操作系统可分为两类，第一类是商用嵌入式操作系统[9]，如微软公司出品的著名系统WinCE。第二类是源代码开放的免费嵌入式操作系统，如应用于卫星发射的ThreadX。虽然现在当今已经出现了众多的嵌入式操作系统，甚至还有一些在某领域取得卓越成绩的，但这并不阻碍本文研究的内容，因为嵌入式操作系统的应用背景相当的广泛[10]。

当今应用较为广泛的嵌入式系统有uC/OS-II，嵌入式LINUX，Windows CE [11]。它们无论从系统结构上，还是从性能参数和功能划分上都存在着较大的差异。

以下对当今应用非常广泛的嵌入式操作系统的任务管理机制做简要的介绍：

（1）uC/OS-II

uC/OS-II是一个可剥夺型的实时多任务内核[12]。在嵌入式领域uC/OS-II凭借其特性得到了广泛的应用[13]。uC/OS-II内部高优先级任务可以剥夺低优先级任务的CPU使用权，处理系统最紧急的事物[14]。uC/OS-II代码可以剪裁，使其变得小巧、灵活、效率高，并且成本降低[15]，最小内核可编译至 2 KB[16] 。uC/OS-II目前已被移植到各

种 8 位、16 位、32 位单片机上[17]。

uC/OS-II提供了专门的延时服务，申请该服务的任务可以延迟一段时间。这段时间的长短是用系统时钟嘀嗒[18]的数目来确定的。实现这个系统服务[19]的函数称为延时函数。调用该函数会使uC/OS-II进行一次任务调度[20] 并且执行下一个优先级最高的就绪态任务。uC/OS-II使用这种方法实现了抢占式调度[21] 。

（2）嵌入式 LINUX

嵌入式 LINUX 是在经典操作系统 LINUX 基础上做了裁剪，取消了虚拟内存管理等模块，适应于代码量精简的嵌入式平台[22]。目前应用比较广泛的LINUX嵌入式操作系统包括: RTAI-LINUX[23]、RED-LINUX[24]、RT-LINUX[25]等等。

（3）Windows CE

Windows CE是一个具有强大通信能力的嵌入式操作系统。它提供了许多快速开发嵌入式系统的工具[26]。Windows CE为开发人员提供了一个相对不错的集成开发环境，开发人员可以在这个环境中完成系统映像的定制、编译和调试[27]。Windows CE具有可靠性好、实时性高等特点[28]。

（4）VxWorks

VxWorks为程序员提供了一些标准API函数，这些API函数集与底层硬件无关[29]。VxWorks为程序员提供了高效的实时多任务调度[30]。VxWorks一般应用于对实时性要求较高的环境，比如美国的火星探测器就是使用VxWorks系统[31]。

（5）QNX

QNX是一种分布式实时操作系统[32]。它的微内核仅提供进程调度、底层网络通信等基本服务服务[33]。QNX广泛应用于自动化、控制、机器人科学及电信等领域[34]。

（6）pSOS

pSOS是一个高性能实时操作系统[35]。 pSOS一般应用于监控系统。其实时内核作为监控软件的核心，对多个异步动作进行协调[36]。

今后，嵌入式操作系统任务管理机制的研究方向主要集中在以下几个方面：

（1）接口函数的标准化，一体化

虽然当今业界已经对嵌入式操作系统的相关规范做出调整。比如美国IEEE协会

在UNIX的基础上，制定了实时UNIX系统接口函数的标准POSIX1001.4 系列协议[37]。但接口函数的标准化工作仍需发展，完善。

（2）GUI、友好界面的完善化

GUI 是用户与嵌入式操作系统的交互层，当今嵌入式系统一般缺乏功能较为完善的 GUI 交互界面，随着应用的背景越来越广泛，应该提供友好，完善的 GUI 界面。

1.3 本文主要研究内容

本文着重研究了当今一些实时操作系统的多任务管理机制，设计和实现了一个专用操作系统 KVM OS 中的任务管理模块。通过设计一套完善的多任务管理机制以增强 KVM 系统的实时性，灵活性，并通过测试平台对 KVM 系统任务管理的性能，效率做出评估。本文具体安排如下：

第一章主要介绍课题的背景，国内外概况。

第二章介绍 KVM 系统任务管理的结构特征，性能标准，相关理论研究及其需求特征。

第三章在分析 KVM 系统任务管理机制的基础上，给出了 KVM 系统任务管理机制的设计方案，包括模型的建立，接口和数据结构的设计等。

第四章详细讨论 KVM 系统任务管理的实现过程。KVM 系统任务管理的实现涉及到很多方面，包括任务调度机制的实现，任务切换机制的实现，定时机制的实现。

第五章对 KVM 系统中的任务管理机制的性能进行测试和分析。

第六章对现有工作的总结和未来研究的展望。

2 KVM OS任务管理简介

本章内容是介绍 KVM 系统任务管理的内容。首先介绍 KVM 系统任务管理的结构特征，其次给出了 KVM 系统任务管理机制的衡量标准，并对每种标准进行了分析和研究，最后给出了 KVM 系统任务管理的需求特征。

2.1 KVM OS任务管理的结构特征

KVM 系统任务管理根据需求分析可以划分为若干模块，分为任务调度管理模块、任务切换管理模块、定时器管理模块三个子模块。其总体结构如图 2.1 所示。

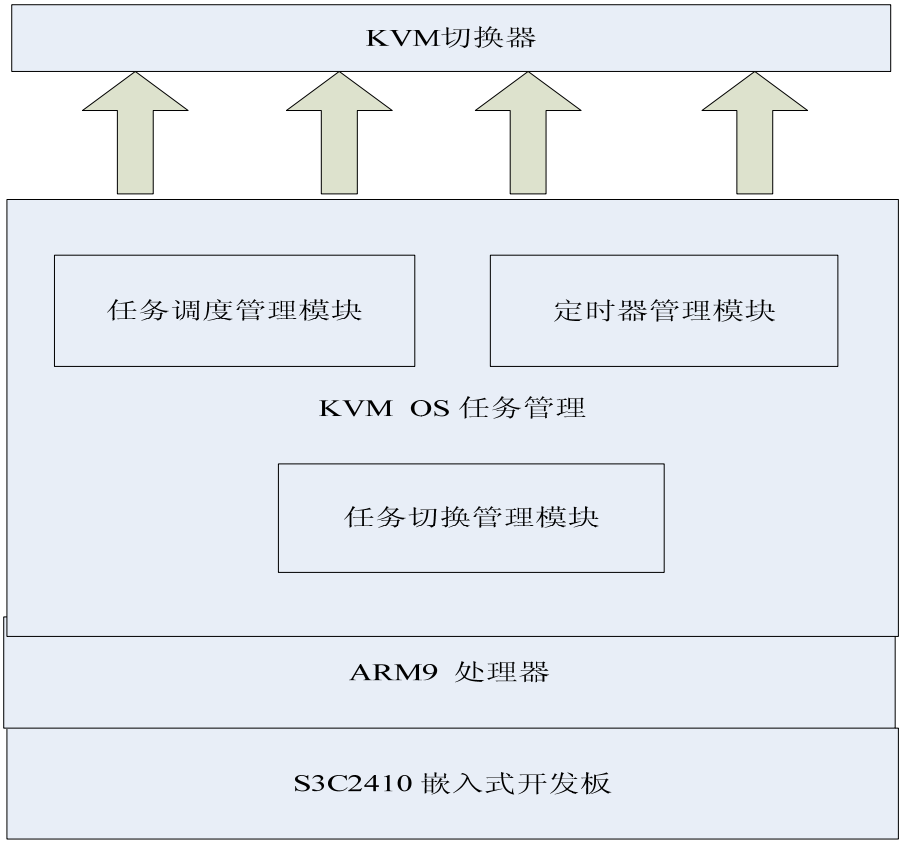

图 2.1 KVM OS 任务管理总体架构

KVM 系统任务管理的各个模块描述如下：

（1）任务调度管理模块

任务调度管理模块对 KVM 系统中的所有任务进行调度，具体包括创建，删除，

挂起，恢复一个任务，以及从所有的就绪态任务中选取一个任务运行。任务调度管理模块最重要的功能是每次调度出一个优先级最高的任务作为即将运行的任务。

（2）任务切换管理模块

任务切换管理模块为 KVM 系统内核的可抢占机制提供了保证。该模块备份当前被打断任务的上下文，然后将任务调度模块调度出的抢占任务恢复到处理器中运行。由于任务切换是 KVM 系统执行最为频繁的操作，所以这个模块应该力求执行速度快。

（3）定时器管理模块

KVM 系统存在大量的定时器。定时器管理模块实现了系统的定时机制。

2.2 KVM OS任务管理的研究内容

用来作为操作系统多任务管理的性能衡量标准主要有以下几种：

1）任务调度(scheduling)：操作系统应该拥有一个调度模块，专门用于从就绪的任务中选择一个合适的任务获得处理器的拥有权。

2）多任务之间的切换[38] (Context-Switch)：在可剥夺型内核的实时操作系统，高优先级任务打断低优先级的任务要靠任务之间的切换来完成。

3）定时器机制(timing)：当今操作系统定义了一类专门用于管理系统时序的特殊任务：定时器任务[39]。

以上性能指标是实时操作系统任务管理机制是否完善的重要衡量标准。下面结合 KVM 系统的特征对各个衡量标准进行研究。

（1）任务调度

任务调度一般指从所有处于就绪态的任务中，以某种策略或算法从多个就需任务中选择一个运行。在抢占式内核中一般调度算法都是基于优先级，即每次调度所有就需任务中优先级最高的那个任务运行。

KVM 系统在任务调度上必须满足调度时间短，同时要保证调度的公平性。时间短是指时间复杂度应该达到 O(1)。公平性指每次应调度优先级最高的任务，如果多个任务优先级相同，则应该以先来先到的方式依次调度这些任务。

（2）任务切换

任务切换，又称为上下文切换。当内核决定运行另外的任务时，它备份被打断任务的当前上下文,即该时刻处理器寄存器数据。然后把即将运行的抢占任务的上下文恢复到处理器中。这个过程叫做任务切换。如图 2.2 所示。

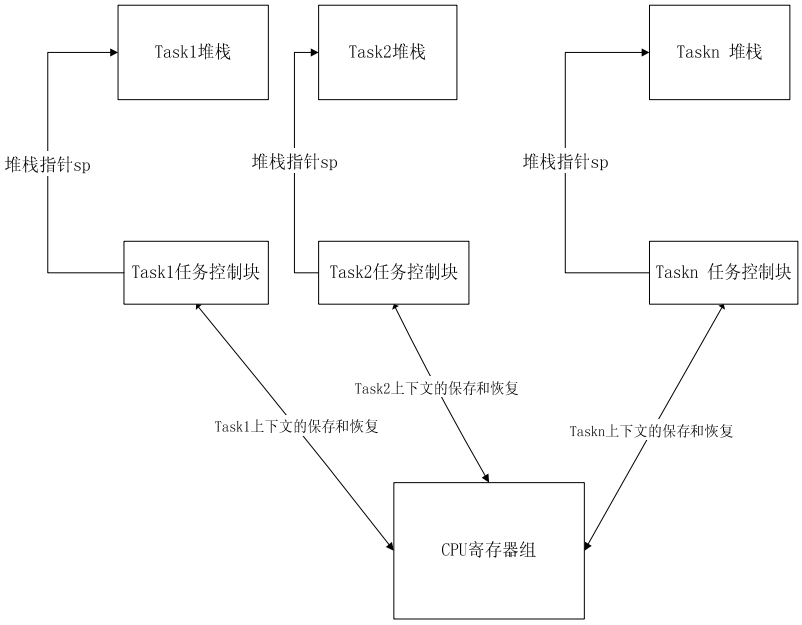

图 2.2　多任务间的上下文切换

任务切换与处理器有直接的关联，处理器的内部寄存器越多，切换需要的时间越长。KVM 系统硬件平台采用 ARM9 处理器，所以有必要对 ARM9 处理器进行研究和分析。

ARM9 处理器包含 37 个寄存器。最多有 18 个活动寄存器。ARM9 处理器为特殊的任务或专门的功能指定了 3 个寄存器：R13,R14 和 R15。

①寄存器 R13 通常作为堆栈指针(SP),保存当前处理器运行模式的堆栈的栈顶地址。

②寄存器 R14 又被成为链接寄存器(LR),保存调用子程序的返回地址。

③寄存器 R15 是程序计数器(PC),其内容是处理器要取的下一条指令的地址。

④CPSR 以及 SPSR 用来监视和控制处理器的状态。

ARM9 处理器处于特殊模式时有 20 个寄存器对程序时隐藏的。这些寄存器被称为分组寄存器。只有当处理器切换到处于某种特定的模式时它们才有效，并且它们

将代替原模式中对应的寄存器。ARM9 中重要的分组寄存器如图 2.3 所示：R13_IRQ；R14_IRQ;SPSR_IRQ。当处理器从用户模式切换成中断模式时，它们将代替R13；R14；CPSR[40,41] 。如图 2.3 所示。

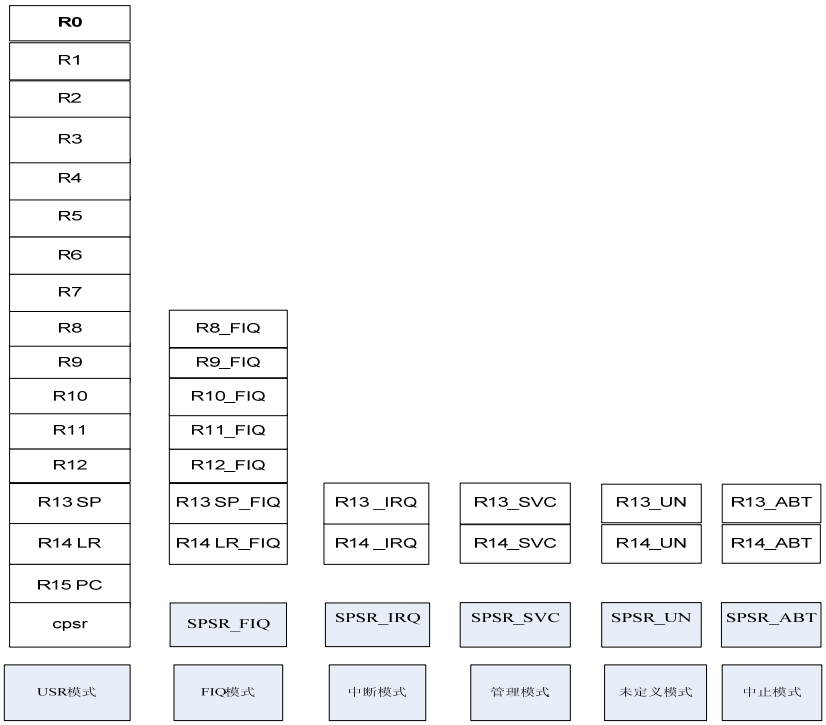

图 2.3　ARM9 不同模式寄存器组示意图

（3）定时器

操作系统通过定时器对时序进行管理。一般的操作系统中都存在一种最为基本的时间度量单位：时钟嘀嗒。时钟嘀嗒由硬件层产生，是最精确的时间度量单位，系统所有的其他时间都是以时钟嘀嗒为基础。KVM 系统定时器任务的时间度量单位也是时钟嘀嗒。定时器从定时功能模式上分为两种：周期定时模式和一次性定时模式。周期性定时器指每隔若干个时钟嘀嗒定时一次。一次性定时器指仅仅定时一次。

2.3　KVM OS任务管理的需求特征

1）实时性

实时性是任务管理的重要需求之一，也是实时操作系统的首要特点。在 KVM 系统任务管理模块中影响实时性的主要因素有：抢占式调度、上下文切换。

（1）抢占式调度

操作系统的实时性在很大程度上由该系统的调度模块的性能决定。为了使权重较高的任务能够及时调度，当今的操作系统一般选择基于优先级的抢占式调度即高优先级的任务能够打断当前较低优先级任务的运行。

（2）上下文切换

上下文切换是 KVM 系统执行频率最高的事件，上下文切换的性能决定了整个系统的实时性。

2）可靠性

KVM 切换器一般应用于大规模服务器的监控，一旦切换器出现故障后果非常严重，所以应该具备较高的可靠性。可靠性是 KVM 系统任务管理机制的又一个重要需求。

2.4 本章小结

本章首先介绍了 KVM 系统任务管理的结构特征，其次对 KVM 系统任务管理的各个子模块进行了分析和研究。研究了当今实时操作系统任务管理机制的各个衡量标准，最后分析了 KVM 系统任务管理的需求特征。

3 KVM OS任务管理的设计

在实时操作系统中，通常是以任务作为资源分配的单位。任务是操作系统中极为重要的概念，应该实现一整套管理机制去描述、管理任务。

KVM 系统将任务分为两种类别：

（1）普通任务

KVM 系统定义了一类执行特定函数的任务，这类任务被赋予优先级，任务调度模块根据优先级对普通任务进行调度。这类任务的所有信息被存储在任务控制块的数据结构中，系统每创建一个普通任务，为该普通任务分配一个任务控制块，接下来多个普通任务相互之间切换，逐个获得处理器使用权。当某个普通任务执行完毕后，被系统删除，其任务控制块被注销掉。

（2）定时器任务

定时器任务是 KVM 系统专门用于定时操作的特殊任务。定时器任务的优先级高于普通任务，当某个时刻存在一个定时器任务需要运行时，调度模块调度该定时器任务，忽略掉当前优先级最高的普通任务。

KVM 系统任务管理主要有以下三方面内容：

（1）任务的调度

KVM 系统中所有的任务都可能经历：创建，挂起，恢复，被选出获得处理器以运行，删除。任务调度管理模块要对以上任务所处的每个过程进行管理。

（2）普通任务的切换

KVM 系统内核属于可剥夺型内核，一个具有更高优先级的普通任务可以打断当前运行的普通任务。所以必须实现一种切换机制，使得抢占任务能够切换掉当前运行的任务。KVM 系统提供了一套普通任务切换机制，实现了系统普通任务之间的切换。

（3）定时器任务的管理

为了满足系统的可靠性，KVM 系统实用了一套定时器任务管理机制，使得系统的普通任务在预期的时延后被定时器任务触发。

在基于多任务的 KVM 系统专用操作系统中，必须对普通任务进行调度。同时应具有提供给应用层的任务管理 API 函数，如创建，删除，挂起，恢复一个普通任务。

3.1 任务状态的定义及转变

在实时操作系统中，任务的执行过程往往具有间断性，这就决定了任务在从创建到删除过程中可能需要经历多种状态。

KVM 系统为普通任务以及定时器任务均提供了一套任务状态的定义。

3.1.1 普通任务的状态及转换

（1）就绪状态

当某个任务获得调度资格时，就立即进入就绪状态。KVM 系统可能出现同一时间存在多个任务处于就绪状态的情况。所有优先级相同的就绪任务排列成一个队列。

（2）挂起状态

当某个任务被挂起函数调用后。其状态被设置为挂起。

（3）删除状态

当任务被删除函数调用后，它的任务控制块中的状态被设置为注销态，以被即将新创建的任务使用。普通任务的状态转变如图 3.1 所示。

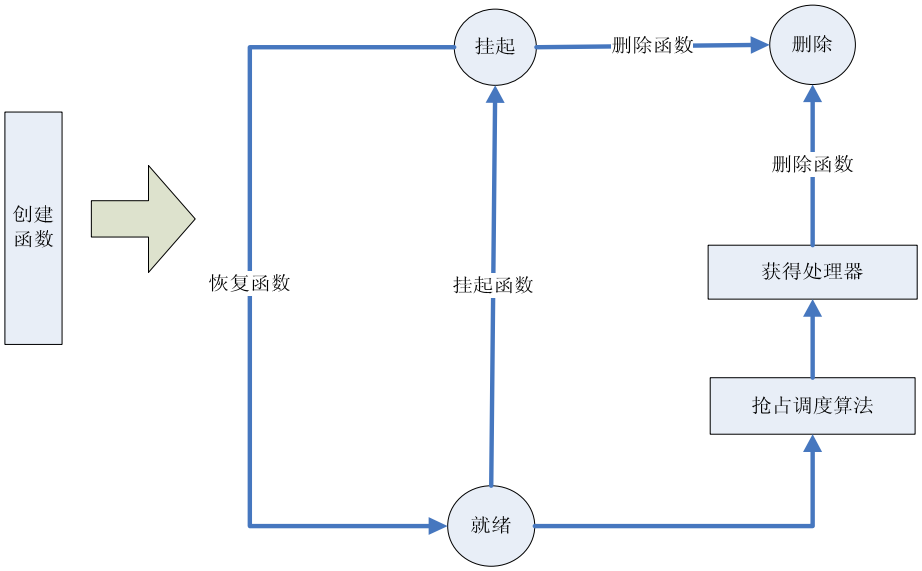

图 3.1 普通任务状态的变迁

3.1.2 定时器任务的状态定义及转换

（1）删除状态

当一个定时器任务不再被使用时，它被系统删除，其任务控制块的状态被设置为删除状态。

（2）就绪状态

当一个定时器任务即将进行计时之前，它的状态将被设置为就绪状态。

（3）挂起状态

当前定时器暂不使用时，将其状态设置为挂起。

3.2 KVM OS任务调度的设计

3.2.1 普通任务的创建与删除

KVM 系统任务控制块设计如表 3.1 所示。

表 3.1 普通任务的任务控制块

名称	含义
索引号 Handle	标识一个任务
堆栈指针 lpSaveSP	指向任务堆栈的栈顶地址的指针
优先级 nPriority	记录任务当前优先级
优先级位图 nPriorityBit	该优先级队列对应的位图数值
就绪队列后续指针 pReadyNextTask	指向下一个就绪任务的指针
就绪队列前驱指针 pReadyPriviousTask	指向当前一个就绪任务的指针
任务状态 nTaskState	记录任务当前的状态

KVM 系统普通任务的堆栈分配由任务创建函数去处理。每创建一个普通任务必须在内存中为其分配任务堆栈空间。本操作系统在内存空间上划分了一片区域给所有普通任务的堆栈。如图 3.3 所示。普通任务的堆栈空间在 Task Stack 范围内。

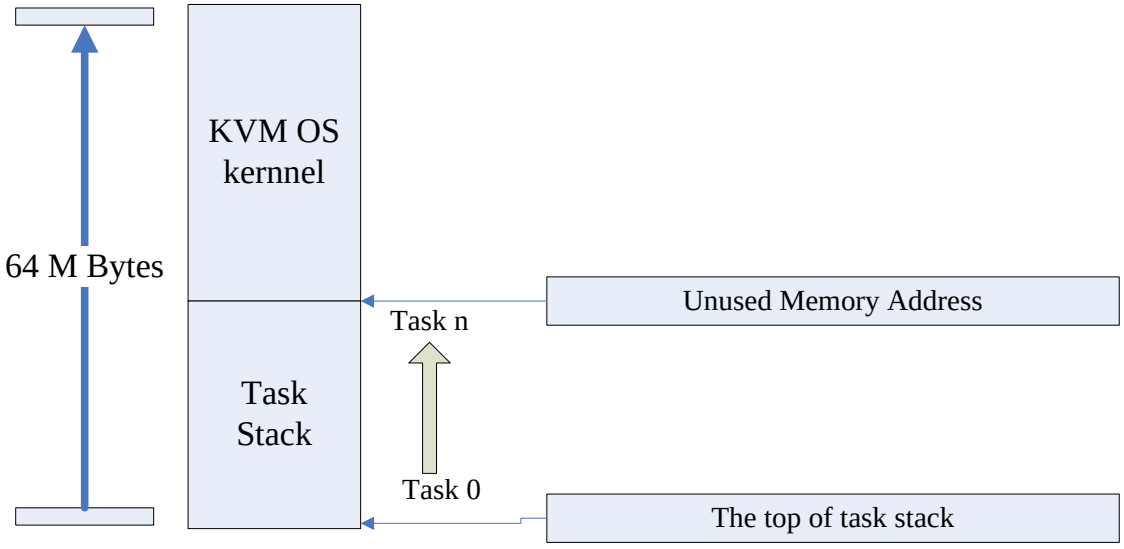

图 3.3　普通任务堆栈分配示意图

系统每创建一个任务，为此任务分配堆栈，将变量 restStackPtr 赋值给该任务的任务控制块中的栈顶指针 lpSaveSP。同时 restStackPtr 的值减去此任务堆栈的空间大小如图 3.4 所示。动态的显示系统还剩多少堆栈空间可以使用。当 restStackPtr 的数值减小到一个数值时表示普通任务的可用堆栈空间已经使用完,应用层无法创建更多的任务。

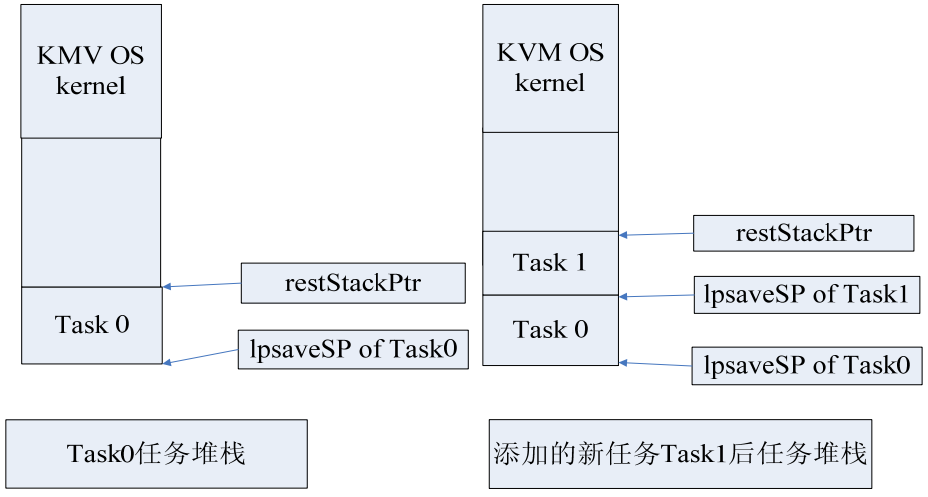

图 3.4　新添加普通任务堆栈示意图

完成堆栈分配和处理器映像的初始化后，就可以将此任务给予调度了，该普通任务的优先级由接口参数传入，调度模块根据此任务的优先级以 FIFO 方式插入到相

关的就绪队列中。

当一个普通任务运行完毕后将被删除。首先注销掉该普通任务的任务控制块。然后清空该普通任务的任务堆栈空间。

3.2.3 普通任务的挂起与恢复

从任务调度的角度来讲挂起一个任务指禁止该任务参与调度，恢复一个任务指允许一个任务参与调度。KVM 系统将所有优先级相同的就绪态普通任务排列在一个队列之中。挂起函数完成的功能是将该普通任务从就绪队列中摘除，同时设置该任务的任务控制块中的状态为挂起态。恢复函数完成的功能是将该普通任务插入到该就绪队列的尾部，同时设置该任务的控制块中的状态为就绪态。

3.2.3 抢占式调度算法

KVM 系统的调度程序使用了基于优先级的抢占式调度和轮转调度相结合的调度策略。将所有的优先级数据存储在一个 BitMap 中，所有优先级相同的任务排列成一个就绪队列如图 3.5 所示。如果某个优先级队列中不存在任何一个就绪任务，则 BitMap 中的相应位为 0，否则为 1。

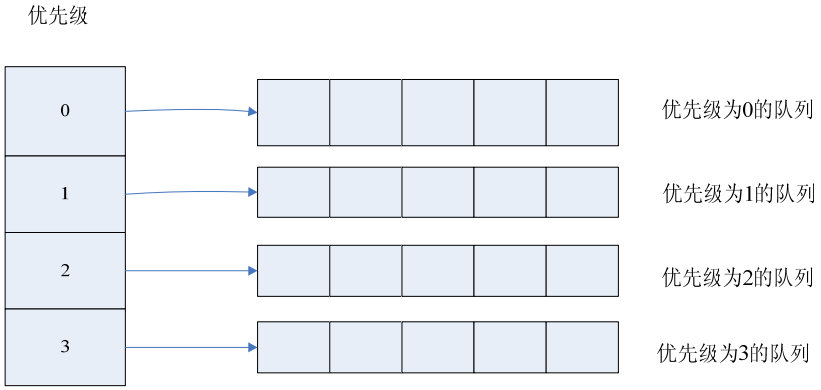

图 3.5 多优先级就绪队列

根据位图查找算法，调度模块每次查找最高优先级的队列，然后按轮转方式依次调度该队列中所有的就绪任务。只有当该队列的所有任务都被调度后，优先级较低的就绪队列才被调度。这样解决了多任务无法具有相同优先级的问题，同时调度算法的时间复杂度仍然为 O(1)。调度模型如图 3.6 所示。

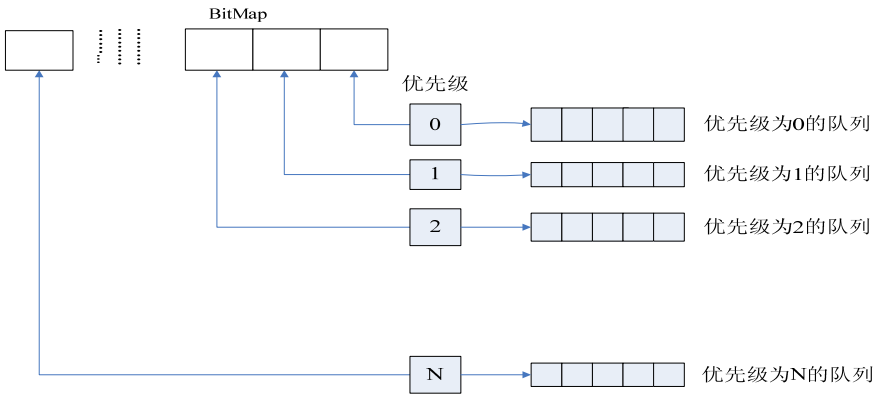

图 3.6　调度模型

3.3　KVM OS普通任务切换的设计

KVM 系统采用 ARM9 处理器，在研究 ARM9 处理器的基础上实现了多任务之间的切换功能。

3.3.1　任务切换模型的设计

上下文切换模块处理当前任务与抢占任务之间的切换。上下文切换完毕后抢占任务就成为当前运行的任务，然后恢复切换之前的上下文环境。模型的设计如图 3.7 所示。

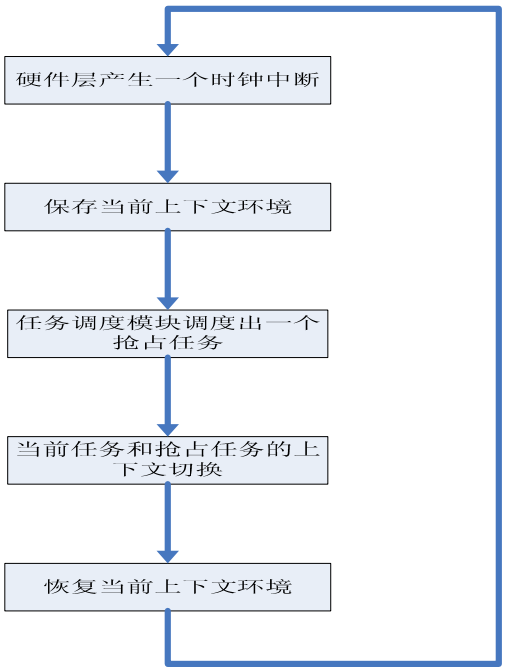

图 3.7　任务切换模型设计图

3.3.2 IRQ堆栈的含义以及上下文切换的设计思想

基于 ARM9 处理器的实时操作系统在内存空间为处理器的每种模式都划分了相应的堆栈空间。当 ARM9 处理器运行于 IRQ 模式时，访问的是 IRQ 堆栈空间。当 ARM9 处理器被一个时钟中断触发后，进入 IRQ 模式，将当前的上下文保存在 IRQ 堆栈中。当完成中断事件后，从 IRQ 堆栈中恢复上下文，使得 ARM9 处理器继续运行被打断之前的事件。

我们将当前运行的任务可以视作被打断的事件，将抢占任务视作打断事件。中断返回后运行的是打断事件是上下文切换的关键。从 IRQ 堆栈的作用可以得出方案：只要将暂存在 IRQ 堆栈中被打断事件的上下文“偷梁换柱”成打断事件的上下文即可。

被打断的任务在 IRQ 堆栈中的上下文即将被覆盖，所以必须将其上下文备份中。我们将每个被打断任务的 IRQ 堆栈中的上下文备份到该任务的任务堆栈中。每个任务堆栈的高 148 字节地址空间用于备份其上下文。这 148 字节地址空间又称处理器映像。

任务堆栈的设计如图 3.8 所示。

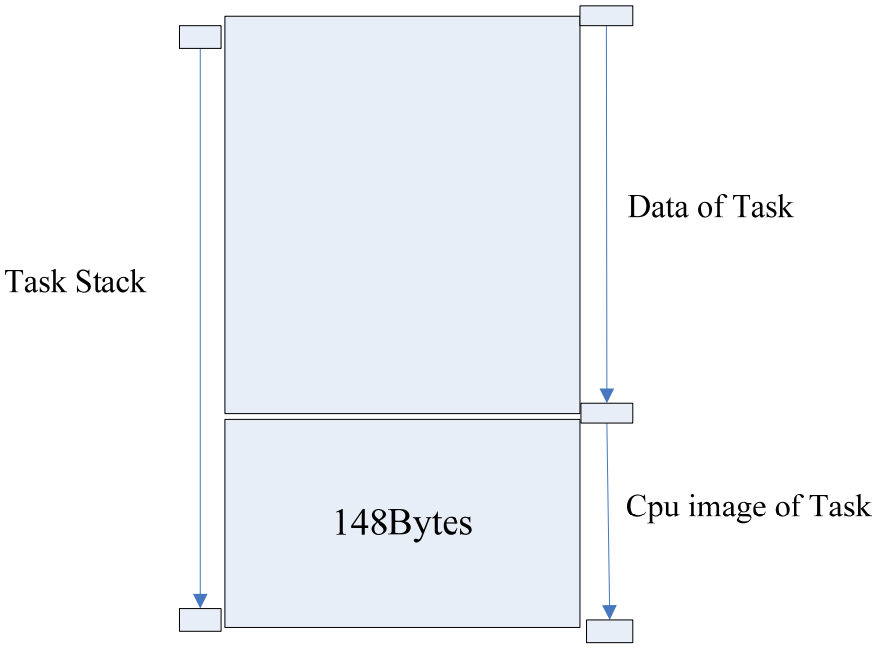

图 3.8　任务堆栈设计示意图

处理器映像是一个数据存储结构，用于备份某一时刻 ARM9 处理器寄存器数据。每个任务通过拷贝处理器映像上的数据到 ARM9 处理器的寄存器上来运行。

在创建一个任务时必须初始化其处理器映像。ARM9 处理器中的各寄存器在处理器映像中的地址排列如图 3.9 所示。

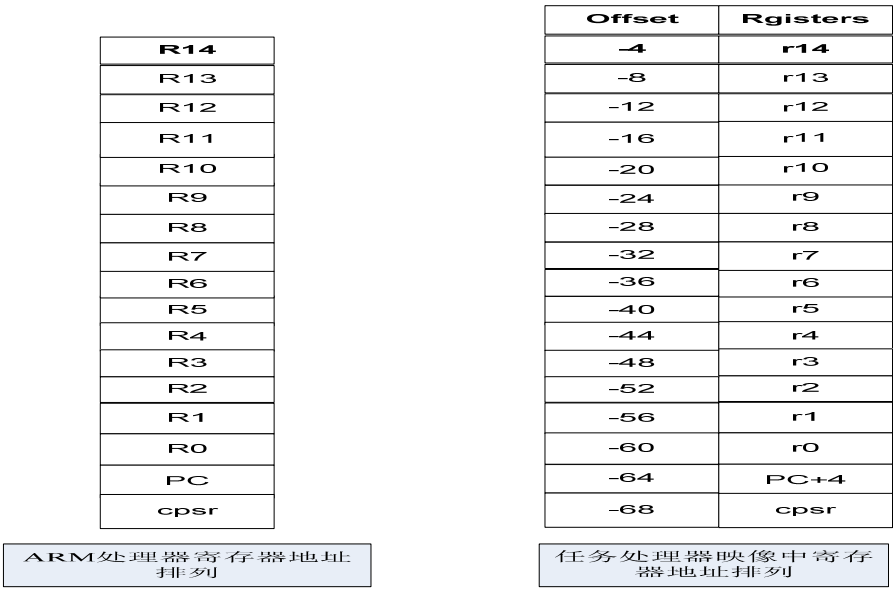

图 3.9　ARM 处理器寄存器与处理器映像关系示意图

在时钟中断到来后。ARM9 处理器进入 IRQ 模式，调度模块产生一个抢占任务。将当前被中断的任务的 IRQ 堆栈中的上下文备份到其任务堆栈中。将抢占任务的任务堆栈中的处理器上下文传递到 IRQ 堆栈中。至此就完成了上下文的切换。待时钟中断返回后，处理器运行抢占任务。设计思想如图 3.10 所示。

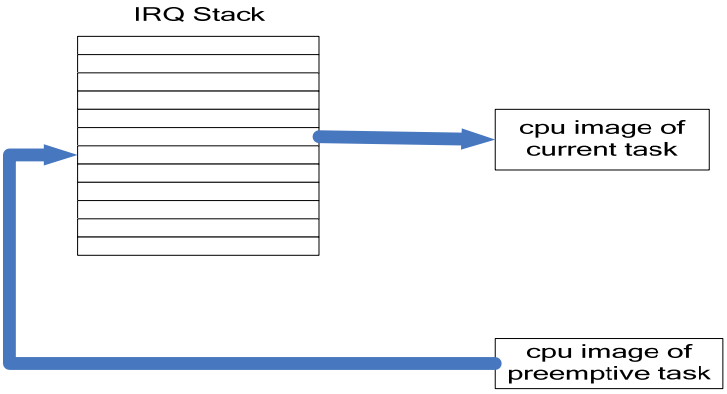

图 3.10　上下文切换中 IRQ 堆栈的处理

3.4 KVM OS定时器任务的设计

3.4.1 定时机制数学模型的建立

在时间管理中存在一个全局变量 iCurTickTimer 表示当前时钟嘀嗒数，我们在定时机制中规定 iCurTickTimer 变量从 0 到一个整数值 TIMER_TICK_NUM 之间循环递增计数。同时存在一个名称为 PtimerTask 的指针数组,其元素的个数为 TIMER_TICK_NUM，每个元素是指向一个定时器任务控制块链表的指针。假设当前时钟嘀嗒数为 T(0<=T<=TIMER_TICK_NUM),我们通过程序设计将当前时刻需要进行操作的所有定时器任务的控制块全部链接在 PtimerTask[T]所指向的链表上。然后处理该链表上所有的定时器任务。如图 3.11 所示。

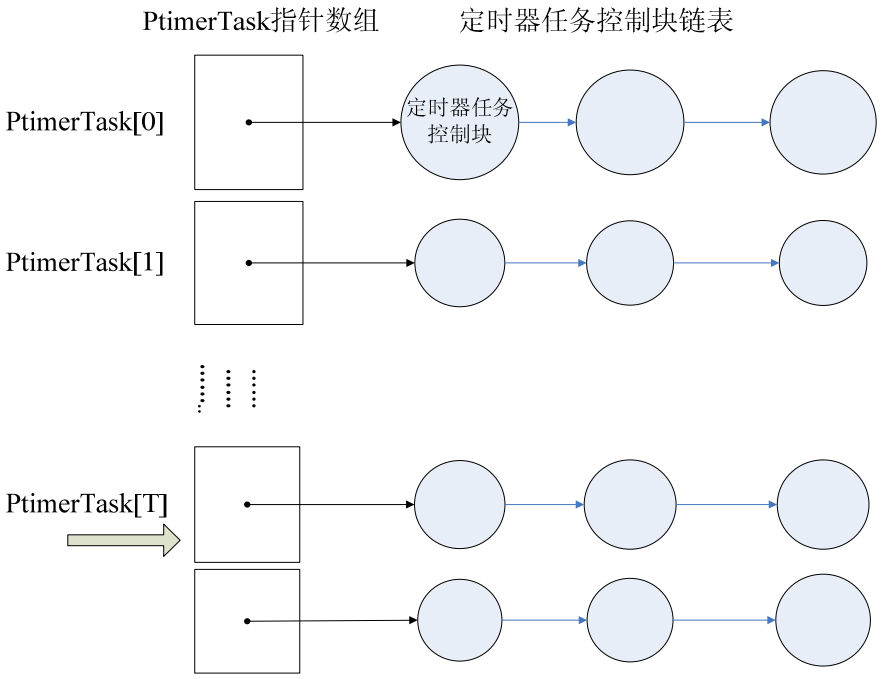

图 3.11 定时机制模型设计示意图

3.4.2 定时器任务控制块的设计

定时器任务的任务控制块中的数据如表 3.2 所示。

表 3.2　定时器任务控制块

名称	含义
定时器任务索引号 ID	用于标识的索引号
挂载链表索引号 list_head	挂载该控制块链表的索引号
指向下一个任务的指针 next	指向链表下一个任务
所处状态 nState	记录当前所处状态
剩余时间 remains_ticks	剩余时间嘀嗒数
周期时间 reschedule_ticks	定时周期
指向执行函数的指针 expiration_function	到期执行的函数
执行函数的参数 expiration_input	到期执行的函数的参数

3.5　本章总结

本章前半部分描述了 KVM 系统任务管理模块的总体结构和一些重要概念，同时描述了任务管理相关的核心数据结构，包括两类任务的任务控制块和一些重要变量。

本章的后半部分具体描述了任务调度模块、普通任务切换模块、定时器模块三个模块的详细设计。每个模块的内容包括模型的建立，整体架构的设计，相关数据结构的设计。

4 KVM OS任务管理的实现

前一章介绍了 KVM 系统任务管理机制的设计。本章将详细讨论任务管理机制的具体实现。KVM 系统任务管理机制的实现涉及到很多方面，本章从任务调度、普通任务切换、定时器任务三个子模块进行论述说明。

4.1 KVM OS任务调度的实现

4.1.1 普通任务的创建与删除

应用层调用 AddTask 添加一个普通任务时，需要指定优先级，堆栈空间等信息，这些信息以函数参数形式传入。

任务创建 API 函数 AddTask( )的执行分为以下几个部分

（1）任务索引号的获取：

遍历普通任务控制块数组，如果发现某个控制块处于注销态，则使用这个控制块。如果没有控制块处于注销态，则返回无控制块可以使用。

（2）任务堆栈空间的分配：

将全局变量 restStackPtr 赋值给该任务堆栈指针 lpSaveSP，然后将 restStackPtr 减去 nStack*1024(该任务所占实际空间为 nStack*1024 bytes),如果结果小于 0，则创建失败，返回空间溢出。

（3）初始化任务消息队列：

将该任务的消息队列设置为空。

（4）初始化任务堆栈中的处理器映像：

处理器映像中有四个寄存器需要初始化，分别为堆栈指示器 R13，链接寄存器 R14，程序计数器 R15，处理器模式状态字 SPSR。根据 ARM9 处理器中寄存器的含义。堆栈指示器 R13 记录该任务数据栈的地址。根据任务堆栈的结构，高 148 字节地址空间为处理器映像，其余地址空间为数据栈，所以应该将 R13 赋值为(lpSaveSP-148)。R14 和 R15 都赋值为指向该任务执行函数的指针。SPSR 赋值为 ARM9 处理器的用户

态模式。

（5）将任务添加到就绪队列：

初始化完成后，该任务可以被调度。调用恢复函数 thread_resume( )将该任务插入到与其优先级相同的就绪队列的队尾。

函数流程图如图 4.1 所示。

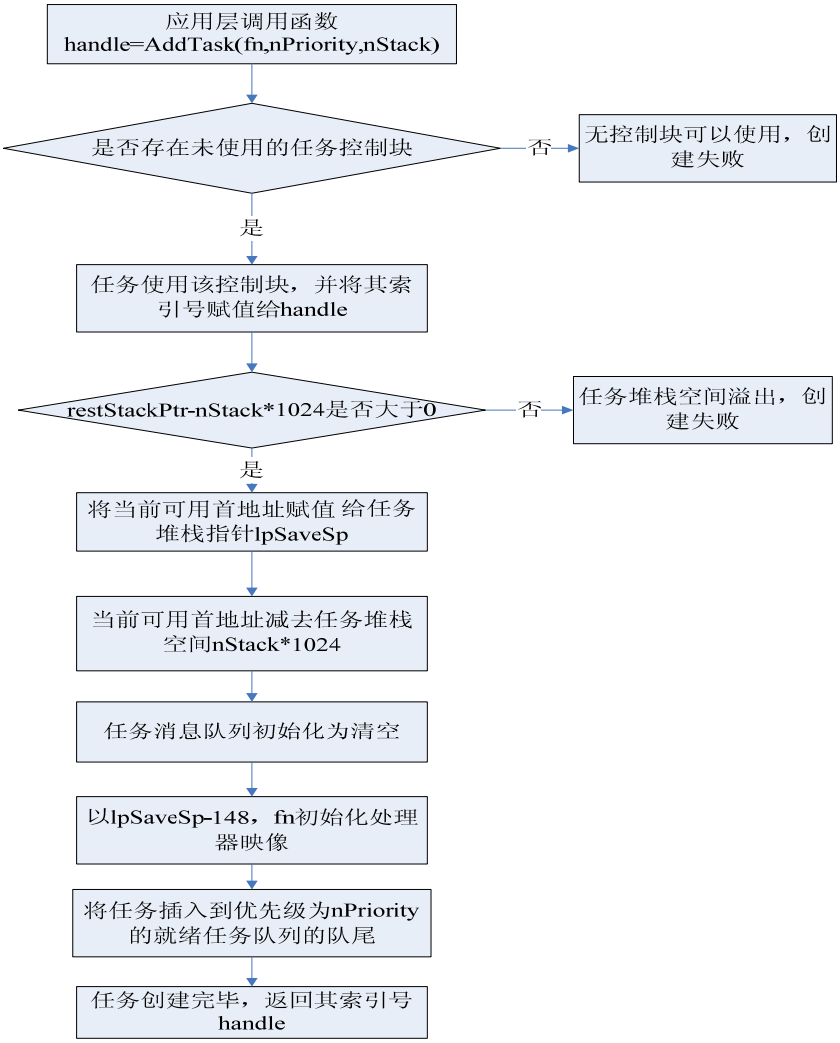

图 4.1　AddTask 函数流程图

任务删除函数的实现较为简单，将该任务的任务控制块状态设置为注销态即可。然后清空该任务的堆栈空间。

4.1.2　普通任务的挂起与恢复

普通任务挂起函数是通过修改任务优先级就绪队列和 BitMap 来实现。假设优先

级为 npriority 的普通任务 task1 被挂起函数调用，如果 task1 任务所在的优先级就绪队列中含有的任务数量大于二，则将 task1 从优先级就绪队列中摘除即可，如果其所在队列仅仅只含此任务，则代表此任务从队列摘除后，队列即将为空。将 BitMap 中 npriority 对应的位设置为空，使得 BitMap 位图查找模块在运行时忽略掉该位。函数的流程图如图 4.2 所示。

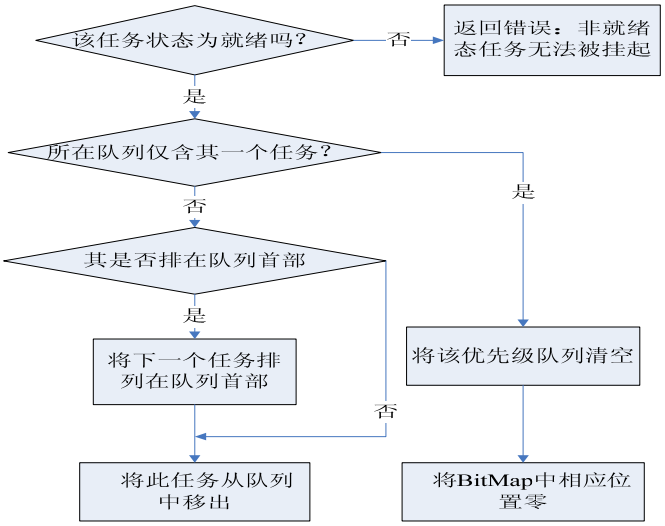

图 4.2　普通任务挂起函数流程图

普通任务恢复函数根据该任务的优先级查看对应的优先级队列情况。如果该优先级就绪队列不为空，则将该任务插入到此队列的队尾。如果该优先级就绪队列为空，则将该任务作为此队列第一个元素。同时设置 BitMap 中该优先级对应的位为 1，使得 BitMap 查找模块在运行时处理此位。恢复函数流程图如图 4.3 所示。

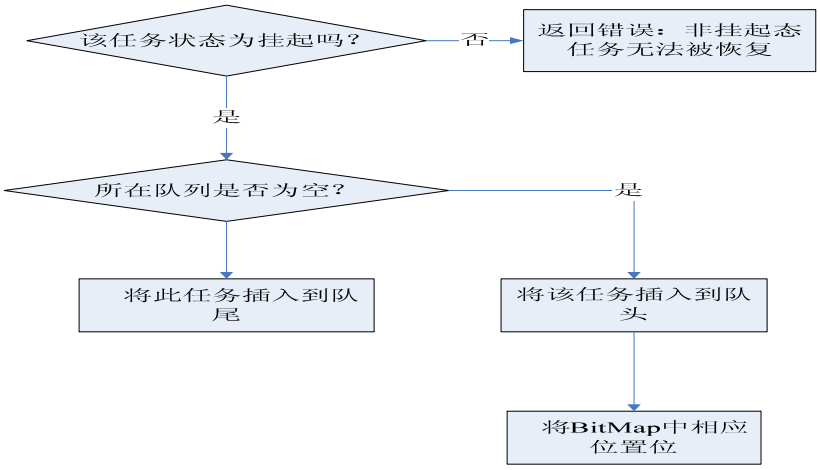

图 4.3　恢复函数流程图

4.1.3 抢占式调度算法

任务调度模块是 KVM 系统执行较为频繁的操作，此模块应力求代码精简，执行速度块。KVM 系统定义了 32 个优先级，数值依次从 0 至 31。数值越小优先级越大。系统存在一个 32 位的位图变量 BitMap，如果某个优先级对应的就绪队列中存在任务，则该优先级在 BitMap 中对应的位为 1，否则为 0。调度模块的功能是查找当前最高的优先级任务，也就是查找 BitMap 中位为 1 的最高优先级。根据 BitMap 的数据结构，最右端对应的是最高优先级 0，最左端为最低优先级 31。

在算法上采用了动态规划的思想。当前 BitMap 中最右端为 1 的位对应的优先级就是当前最高优先级。为了不破坏系统的 BitMap，复制出一个当前系统 BitMap 的副本 BitMap1。对 BitMap1 进行位运算处理。为 BitMap1 构造五个掩码如表 4.1 所示。

表 4.1 BitMap1 掩码图

名称	移位	数值	作用
M0	16	0x0000ffff	当前 BitMap1 0-15 位是否存在某位为 1
M1	8	0x00ff0000	当前 BitMap1 16-23 位是否存在某位为 1
M2	4	0x0f000000	当前 BitMap1 24-27 位是否存在某位为 1
M3	2	0x30000000	当前 BitMap1 28-29 位是否存在某位为 1
M4	1	0x40000000	当前 BitMap1 30 位是否为 1

以上五个掩码将 BitMap1 划分为连续的五个模块，如图 4.5 所示。现假设当前 BitMap1 中最高优先级对应的位属于某个模块，此时应该将 BitMap1 向左移动该模块对应的位数。然后判断此时该位移动到了哪个模块，类似的向左移动相应位。直到该位移动到了 BitMap 的最左端。最低优先级数值为 31，每次移动时将数值 31 递减移动的位数数值。最后移动到最左端时，递减后的剩余数值就是该位对应的优先级数值。

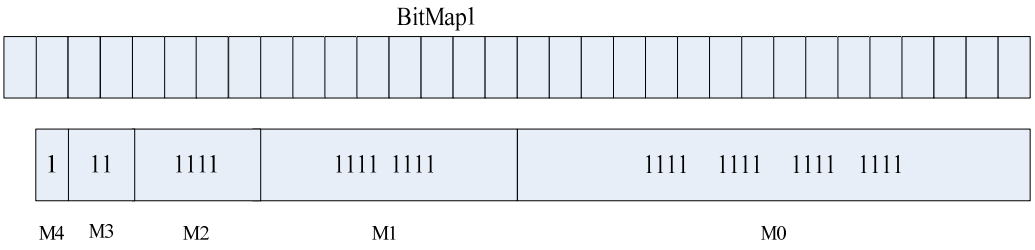

图 4.5　BitMap1 的划分

在实现上首先处理的是 M0 掩码对应的模块，因为 M0 的位数较低，对应的优先级较高。如果 M0 中不存在任何一位为 1，再处理 M1 模块，类似直到 M4。查找到该优先级后，将此优先级对应的就绪队列中排在对头的任务作为即将调度的任务。至此就完成了调度操作。

程序设计的核心代码如下：

```
 highest_priority = 31;
if (priority_map & 0x0000ffff)
{ highest_priority -= 16;
   priority_map <<= 16; }
if (priority_map & 0x00ff0000)
{ highest_priority -= 8;
   priority_map <<= 8;   }
if (priority_map & 0x0f000000)
{ highest_priority -= 4;
   priority_map <<= 4;   }
if (priority_map & 0x30000000)
{ highest_priority -= 2;
   priority_map <<= 2;   }
if (priority_map & 0x40000000)
{ highest_priority -= 1; }
```

4.2　KVM OS普通任务切换的实现

上下文切换是 KVM 系统执行最为频繁的操作，为了达到执行速度快，该部分采用 ARM9 汇编语言编写。

4.2.1 IRQ堆栈与任务堆栈处理器映像中寄存器的地址分配

保存和恢复寄存器数据所消耗的时间与寄存器的数量成正比。为了缩短上下文切换执行的时间，仅仅备份若干重要的寄存器数据。IRQ 堆栈中的寄存器排列顺序如图 4.6 所示。

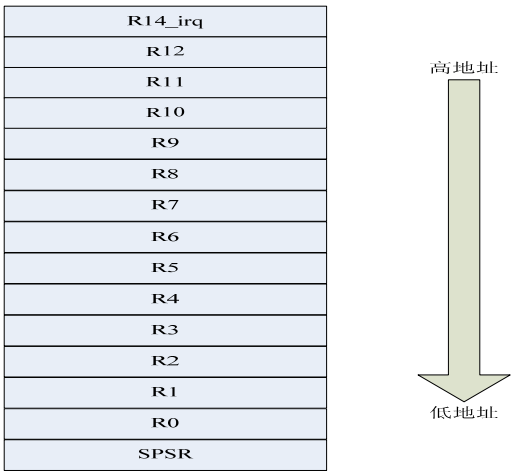

图 4.6 IRQ 堆栈中处理器映像中的寄存器排列图

当某个任务在被切换之前，处理器处于用户模式，进入 IRQ 模式后必须备份用户模式下的寄存器到该任务堆栈中。依据 ARM9 处理器寄存器的特征，将每个任务堆栈中的寄存器排列设置为：高 60 字节地址空间用于存储 ARM9 处理器用户模式下的寄存器 R14-R0，其余的 8 字节地址空间用于存储 ARM9 处理器 IRQ 模式下特有的寄存器 R14_irq, SPSR。

实际上 IRQ 堆栈上某个任务的处理器映像是该任务堆栈中处理器映像的一个子集。任务堆栈中处理器映像中寄存器排列如图 4.7 所示。

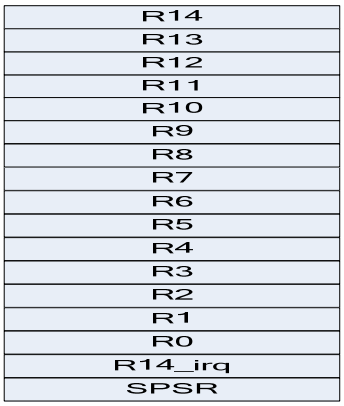

图 4.7 任务堆栈中处理器映像中的寄存器排列图

上下文切换实现的关键是 IRQ 堆栈上处理器映像的替换,即抢占任务的处理器映像替换掉当前被打断任务的映像。

4.2.2 利用ARM9 处理器中转传送的方案

将 IRQ 堆栈中某任务的处理器映像备份到该任务堆栈时，不是直接进行拷贝，而是利用 ARM9 处理器作为中转传递的方案：将 IRQ 堆栈的处理器映像中的寄存器数据传递到 ARM9 处理器，再将 ARM9 处理器再将寄存器数据传递到任务堆栈中的处理器映像。采用这种方案的理论依据如下：

IRQ 堆栈和任务堆栈均属于内存空间，虽然采用中转传递要经过两次寄存器和内存之间的拷贝，但这两次传递消耗的时间必定远小于一次内存和内存之间的拷贝，因为内存和内存之间传递数据消耗的时间要比寄存器和内存之间传递数据的时间高出几个数量级。

4.2.3 上下文切换函数context_switch( )的具体实现

在进入函数后首先将当前处理器环境备份在 IRQ 堆栈的一片地址空间。我们将这段地址空间命名为函数现场空间。IRQ 堆栈中另外有一段 60 字节的地址空间用于存储任务处理器映像。交换此地址空间的处理器映像从而实现了任务的切换。进入切换函数时，处理器当前环境压入堆栈指针 SP 指向的函数现场空间。此后 SP 定位到该任务的 IRQ 堆栈处理器映像。将该空间的所有寄存器数据传递到 ARM9 处理器。然后 ARM9 处理器再将这些寄存器数据传递到该任务堆栈。到此完成了当前任务 IRQ 堆栈处理器映像的备份操作。如图 4.8 所示。

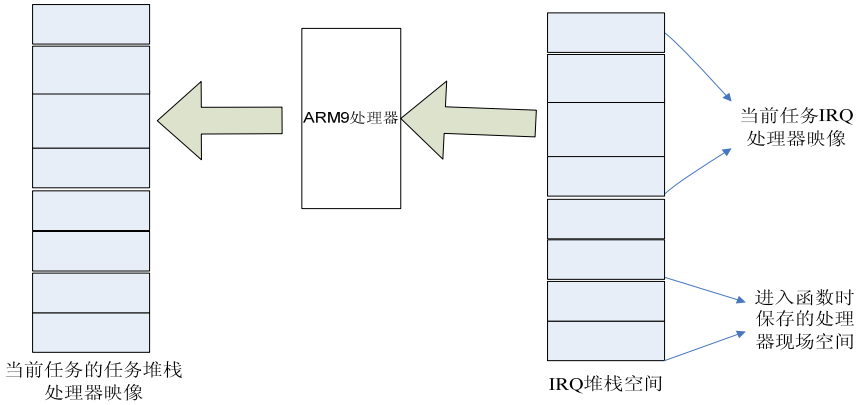

图 4.8 当前任务处理器映像的备份

当前任务处理器映像的备份操作完毕后，将抢占任务的任务堆栈中的处理器映像传递至 ARM9 处理器，然后再传送到 IRQ 堆栈交换空间。实现了 IRQ 堆栈中处理器映像的切换。如图 4.9 所示。

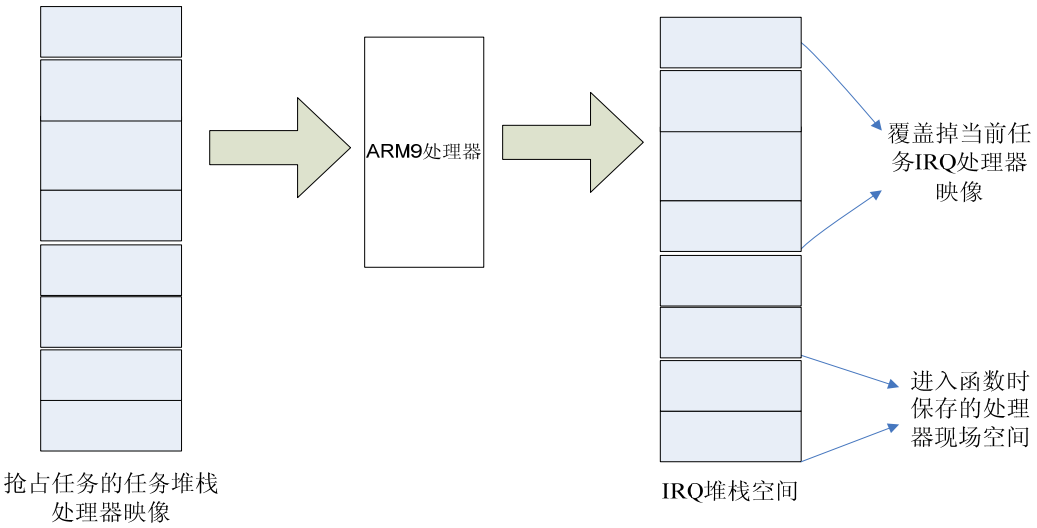

图 4.9　抢占任务处理器映像的恢复

最后将进入上下文切换函数时备份的处理器现场恢复，使得函数能够返回。

4.3　KVM OS定时器任务的实现

4.3.1　时序间隔与定时器任务控制块链表索引号的映射

如何将现实中的时钟嘀嗒数的增加转换成对 PtimerTask 数组中不同索引号链表的访问是本操作系统中定时功能实现的关键。

如图 4.10 所示，假设当前的时钟滴嗒数为 T (0<=T<=TIMER_TICK_NUM)，定时器函数内部访问 PtimerTask[T]指向的定时器任务控制块链表。这个链表中所有的定时器任务在时间逻辑上分为两类。第一类：它们是在 T 时刻刚刚到时。第二类：它们距离到期时间的时钟滴嗒数大于 TIMER_TICK_NUM。遍历当前链表中所有的定时器任务，如果属于第一类，立即执行定时器到期函数。如果属于第二类，将该定时器的剩余时间减去 TIMER_TICK_NUM。时钟嘀嗒数增 1，接下来访问 PtimerTask[T+1] 。依次直到访问 PtimerTask 数组中最后一个元素

PtimerTask[TIMER_TICK_NUM]。当 PtimerTask[TIMER_TICK_NUM]指向的定时器任务控制链表访问完成后，表示此次循环定时结束。接下来重新从 PtimerTask[0]开始访问。

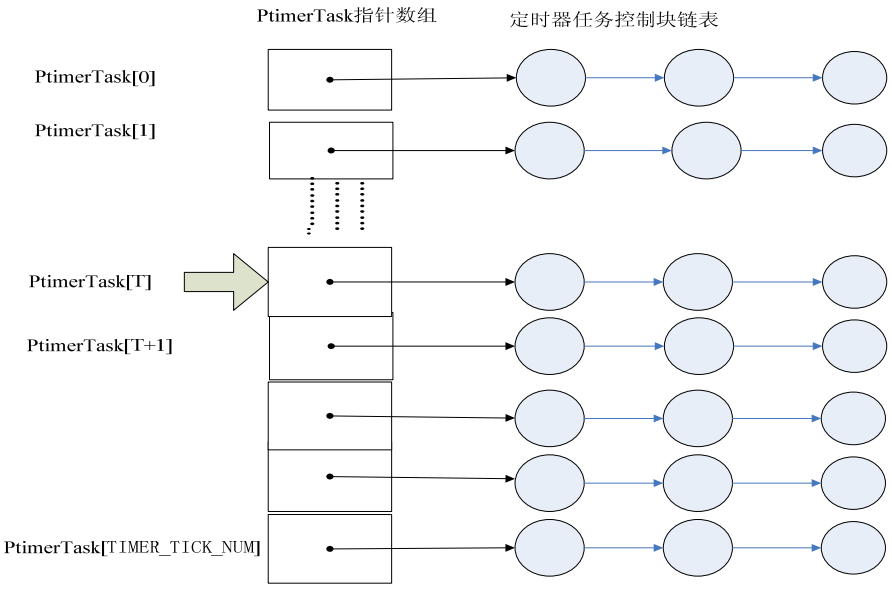

图 4.10　时钟嘀嗒的增加与对 PtimerTask 不同索引号链表的访问

4.3.2　定时器任务控制块链表插入算法

假设当前的时钟滴嗒数 iCurTickTimer 为 T(0<=T<=TIMER_TICK_NUM)的时刻，系统启动一个定时器任务开始计时，从此时刻距离到期时刻的时钟滴嗒数为 remains_ticks。

应该根据 remains_ticks 的数值将此定时器任务控制块插入到 PtimerTask 中某个链表。分为以下三种情况

（1）remains_ticks<(TIMER_TICK_NUM-T)

(TIMER_TICK_NUM-T)表示从此时开始到此次定时周期结束之间的时钟滴嗒数。如果remains_ticks值较小，在这次系统定时周期结束之前这个定时器任务就会到时。只需将此定时器任务控制块添加到PtimerTask[T+remains_ticks]所指向的链表尾部。从此时刻起再过 remains_ticks 个时钟嘀嗒数后，即将遍历 PtimerTask[T+remains_ticks]指向的链表。这样就实现了再过remains_ticks个时钟嘀嗒

后这个新启动的定时器任务到时。如图4.11所示。

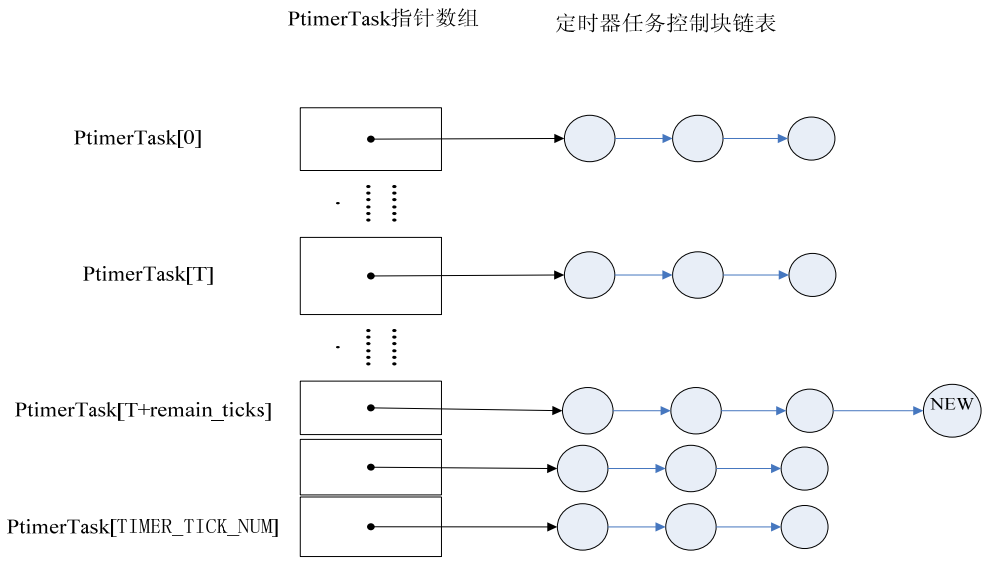

图 4.11　新启动的定时器任务插入合适的链表

（2）(TIMER_TICK_NUM-T)< remains_ticks < TIMER_TICK_NUM

如果新启动的定时器剩余时钟嘀嗒数较大，在此次时钟嘀嗒周期循环结束还没有到时，iCurTickTimer 从当前的 T 增长到 TIMER_TICK_NUM 后会变为 0 重新递增。由于 remains_ticks 小于 TIMER_TICK_NUM，即在下一个时钟嘀嗒周期之内必定会运行。

假设应该将此定时器的任务控制块插入到 PtimerTask 数组中索引号为 T1 的链表。当前距离到此次周期定时结束的时钟嘀嗒数为 TIMER_TICK_NUM-T。下一个时钟嘀嗒周期内遍历到 T1 号链表时此定时器任务到时。固有以下等式成立：

TIMER_TICK_NUM-T+T1= remains_ticks

T1= remains_ticks+T- TIMER_TICK_NUM

所以这种情况下应该将新启动的定时器任务控制块插入(remains_ticks+T-TIMER_TICK_NUM)号的链表中。

（3）TIMER_TICK_NUM <remain_ticks

对于这种情况，当前启动的定时器的到时期限很长，在下一个时钟嘀嗒周期内仍不可能到时。将其任务控制块插入当前链表的尾部，在下一个时钟嘀嗒周期遍历到 PtimerTask[T]时，说明经历了 TIMER_TICK_NUM 个时钟嘀嗒，将此定时器任务

控制块的 remains_ticks 数值减去 TIMER_TICK_NUM，然后比较 remain_ticks 与 TIMER_TICK_NUM 的大小。

如果 remain_ticks 仍然大于 TIMER_TICK_NUM，说明在下一个时钟周期内还是没到期，重复以上操作。否则说明在此周期内或是下一个周期内定时器必然到期，即此时属于第一种情况或是第二种情况，将此定时器任务控制块插入到相关的链表的尾部。如图 4.12 所示。

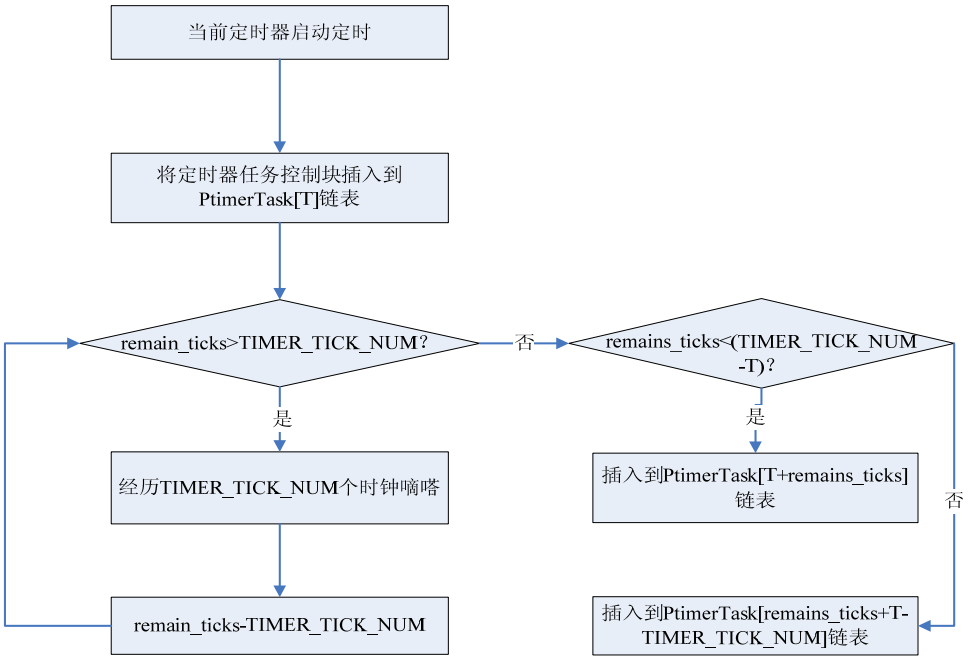

图 4.12　定时时间较长的新添加定时器任务插入链表流程图

4.3.3　具体API函数

定时器定时功能由具体的 API 函数实现，这些函数随着系统时钟嘀嗒的变化对定时器控制块链表进行动态的操作。实现定时器定时器功能的主要 API 函数包括 insert_timer( )和 Timer_Task( )。这两个函数的实现过程如下：

（1）启动定时函数 insert_timer( )

此函数入口参数是一个刚启动的定时器任务控制块索引号，函数根据控制块得到其剩余时间 remains_ticks，然后根据 remains_ticks 的值，依照定时器任务控制块链表插入算法得到待插入链表头指针的序号。最后将此定时器任务控制块插入到此链

表的尾部。实现流程如图 4.13 所示。

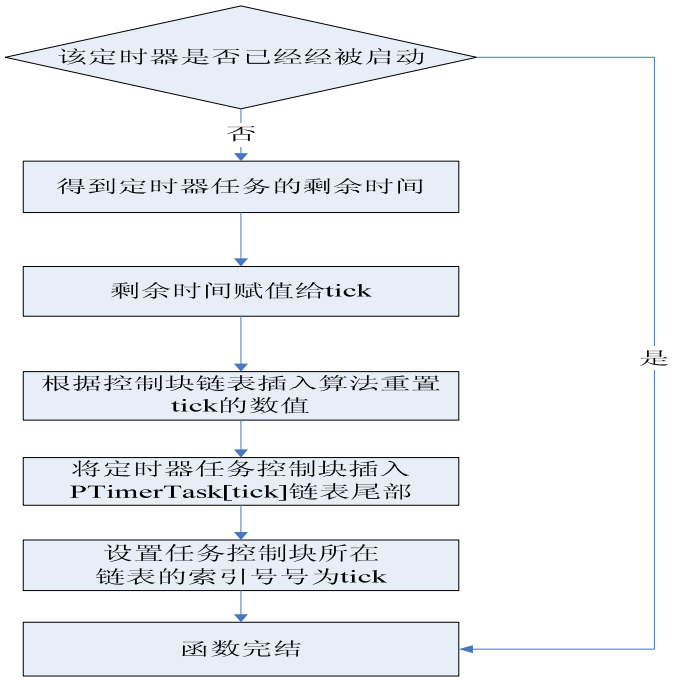

图 4.13　启动函数 insert_timer( )流程图

（2）计时功能函数 Timer_Task( )

Timer_Task( )是定时功能中的核心函数。函数从 0 号链表一直扫描到最后一个链表，然后循环再次扫描。假设当前时刻的时钟嘀嗒为 T，将 PTimerTask[T]指向的定时器任务控制块链表摘除。从该链表的第一个任务控制块开始逐个处理，直至链表尾部。对于每一个定时器任务控制块，首先将其从链表上摘除，然后判断其状态，如果不是就绪状态则忽略掉。如果其剩余时间 remains_ticks 小于 TIMER_TICK_NUM，根据定时器任务控制块链表插入算法说明此定时器在此时刻到时，将其任务索引号赋值给当前定时器任务 iCurTimer，执行到期函数。

如果该定时器是一个周期性定时器，将它的定时周期的时钟嘀嗒数 reschedule_ticks 赋值给剩余时间 remains_ticks，然后对此定时器调用定时启动功能函数 insert_timer( )，距此时刻 remains_ticks 个时钟嘀嗒后再次到时，以实现其周期定时功能。如果该定时器是单次定时，那么在其运行完到期函数后将其状态设置为挂起以取消其定时功能。如果 remains_ticks 大于 TIMER_TICK_NUM，根据定时器任务控制块链表插入算法，说明对 PTimerTask 的 TIMER_TICK_NUM 个链表进行了一

次周期遍历又回到了 PTimerTask[T]处。将其 remains_ticks 减去 TIMER_TICK_NUM。插入到合适的链表中。实现流程图如图 4.14 所示。

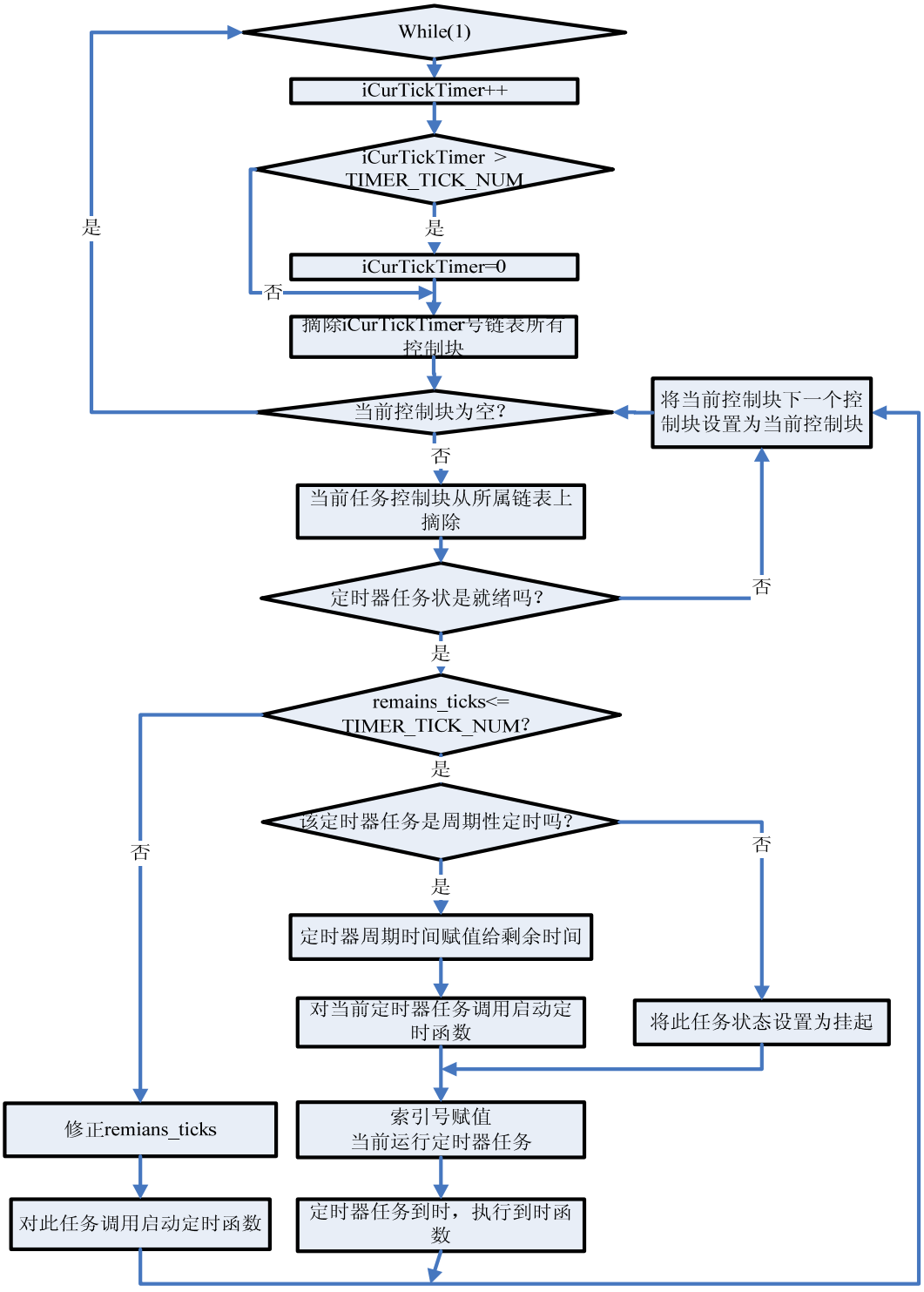

图 4.14　Timer_Task( )流程图

4.4 本章小结

本章论述了 KVM 系统任务管理机制的具体实现。任务调度的实现给出了具体的任务调度算法描述，详细描述了任务调度函数的程序流程。普通任务切换的实现重点论述了任务的处理器映像的传递轨迹。定时器任务的实现着重描述了对定时器控制块链表的操作流程。

5 测试与分析

本章主要内容讲述了 KVM 系统任务管理的性能测试指标；以及测试平台的搭建，包括软件、硬件环境配置。给出了测试案例的设计以及图表化的测试数据。最后将 KVM 系统任务管理的性能测试数据与当今若干嵌入式实时操作系统的对应部分做了比较。

5.1 测试环境

KVM 系统是一个运行在 ARM 嵌入式硬件平台的操作系统。在 ARM 平台，一般使用集成的 ADS 开发环境配合 ARM JTAG 仿真器是目前采用较多的测试方式，它可以满足不同用户对嵌入式系统开发的要求。

基于 JTAG 仿真器的 ADS 测试环境的基本结构包括测试主机，协议转换器和测试目标。

测试主机一般是一台运行测试软件的计算机。本项目采用操作系统为 windows XP 的 PC 机，在该 PC 机安装两套工具：软件开发工具 Code Warrior IDE 和软件调试工具 AXD。

软件开发工具 Code Warrior IDE 是 ARM 平台流行的集成开发环境，集合了 ARM 汇编器、ARM/Thumb 的 C/C++编译器、ARM 链接器。完成代码编辑，编译，链接整个流程。

软件调试工具 AXD 的全称是 ARM eXtended Debugger,即 ARM 扩展调试器。具体功能包含：把需要调试的可执行映像文件装载到目标板上。启动和停止目标板上的程序。显示或修改内存、寄存器和程序变量的数值。

协议转换器属于调试代理的一部分。调试代理的作用有两个。一执行测试主机的调试软件 AXD 发出的高层调试命令(如设置断点)。二协议转换。通过并口协议和测试主机通信，使用 JTAG 协议和目标板通信。本项目采用 MULTI-ICE 作为调试代理。

调试目标一般有两类：硬件目标板和模拟软件。硬件目标板指基于 ARM 系列内

核的目标开发板。模拟软件指模拟 ARM 处理器的软件模拟器。本项目采用硬件目标板，选择基于 ARM9 处理器的 S3C2410 目标板。

测试环境的搭建步骤如下：

（1）将 Multi-ICE 通过 JTAG 连接到 S3C2410 目标板上。

（2）在测试 PC 机运行 Multi-ICE 服务器软件。

（3）在 Code Warrior IDE 编译 KVM 系统代码，生成最终的可执行映像 axf 文件。

（4）通过 JTAG 将 axf 文件烧写到 S3C2410 目标板上。

（5）将 S3C2410 目标板上的串口与测试 PC 机上的串口之间连接上串口线。打开测试 PC 机的超级终端并设置属性。

（6）运行 KVM 系统，记录超级终端上出现的数据。

5.2 性能测试分析

5.2.1 普通任务切换性能测试分析

首先测试 KVM 系统任务切换机制的正确性，在满足切换机制正确性的情况下，测试多任务情况下，切换平均所需时间。

（1）任务切换测试案例的设计

首先测试切换机制的正确性，创建两个优先级一样的任务分别 Task1,Task2。为了测试的方便两个任务执行的指令较为简单，仅仅是打印一行语句。运行结果如图 5.1 所示。

在正确性得到验证的情况下测试多任务切换平均所需时间。选择任务数量分别为 60，120，180，240，300 的测试集。

每个测试集测试三次，每次测试如下。

第一次设置所有的任务优先级均相同。

第二次设置一半数量的任务优先级为 20，另外一半数量的任务优先级为 40。

第三次设置三分之一数量的任务优先级为 20，三分之一数量的任务优先级为 40，三分之一数量的任务优先级为 60。

测试方法采用选取一段制定长度的时间，然后统计该时间内发生切换的所有任

务的数量，最后用时间除以任务数量得出每个任务的平均切换时间。

以上三个测试集的每次测试的任务切换平均时间如表 5.1 所示。

（2）测试结果及分析

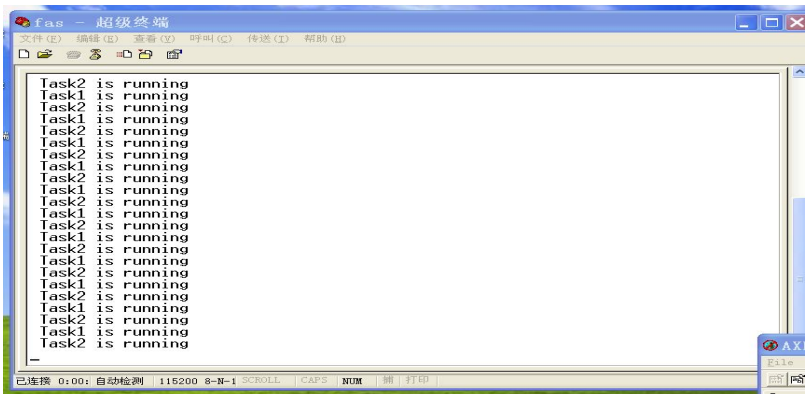

图 5.1　两个优先级相同任务间切换运行

由图 5.1 可知 Task1 和 Task2 两个任务相互之间切换运行。 说明了 KVM 系统任务切换机制正确。

表 5.1　各测试集任务切换平均时间

序号	任务数量	各优先级任务数量比例	任务切换时间(us)
1	60	所有任务相同	7.09
2	60	20，40 数量 1:1	7.11
3	60	20，40，60 数量 1:1:1	7.13
4	120	所有任务相同	7.10
5	120	20，40 数量 1:1	7.12
6	120	20，40，60 数量 1:1:1	7.13
7	180	所有任务相同	7.10
8	180	20，40 数量 1:1	7.13
9	180	20，40，60 数量 1:1:1	7.14
10	240	所有任务相同	7.11
11	240	20，40 数量 1:1	7.13
12	240	20，40，60 数量 1:1:1	7.15
13	300	所有任务相同	7.12
14	300	20，40 数量 1:1	7.14
15	300	20，40，60 数量 1:1:1	7.16

根据表 5.1 所记录的切换时间数据，任务切换时间与任务数量的关系如图 5.2 所示。

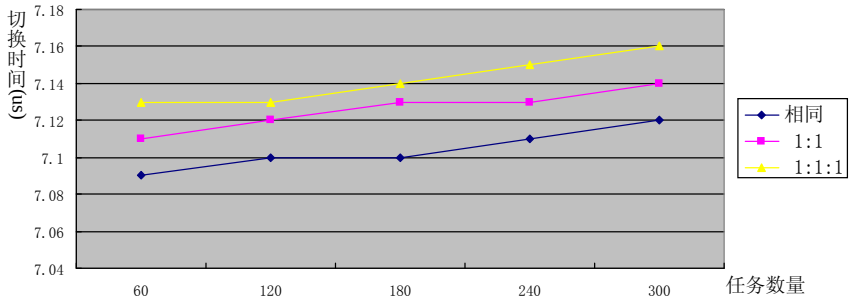

图 5.2　任务切换时间与任务数量对比图

由图 5.2 可知，随着系统任务数量的增多，任务切换消耗的平均时间呈递增趋势。同时任务彼此之间的优先级差别越大，系统的平均任务切换时间也越长。所有任务优先级相同时，平均任务切换时间最短。

由图 5.2 可以得出 KVM 系统的保守任务切换平均时间为 7.14us。现列举一些当今著名商用是实时操作系统的任务切换时间。在以下的嵌入式操作系统中，RTA-OSEK 应用于航空系统；LynxOS、Nucleus PLUS 以及 QNX 代表了传统的主流实时操作系统。这些实时操作系统的任务切换平均时间(us)如表 5.1 所示。

表 5.1　著名商用实时操作系统任务切换平均时间

名称	任务切换平均时间(us)
RTA-OSEK	5
QNX	6.5
LynxOS	15
Nucleus PLUS	30

5.2.2　定时器性能测试分析

定时器的功能是距离此时刻一定时期后触发一个任务，首先测试 KVM 系统定时器功能的正确性，在满足功能正确性的前提下再测试定时器定时的精确性。

（1）定时器测试案例的设计

创建定时器任务的 API 函数形式为：

```
UINT     _txe_timer_create(TX_TIMER *timer_ptr, CHAR *name_ptr,
VOID (*expiration_function)(ULONG), ULONG expiration_input,
ULONG remains_ticks, ULONG reschedule_ticks, UINT auto_activate);
```

该函数接受五个用于设置定时器任务属性的参数。第一个参数为指向定时器任务的指针，第二个参数为定时器名称，第三个参数为触发的普通任务，第四个参数是该任务的参数，第五个参数是从创建该定时器任务开始，多少时钟嘀嗒后触发任务，第五个参数用于周期性定时器，即多少时钟嘀嗒该定时器触发任务一次。最后一个参数是该定时器任务被创建后是否立即开始计时。首先创建一个普通的任务 Task1,为了测试方便 Task1 仅仅打印出一条语句。然后创建一个定时器任务 Timer1，设置从 200 时钟嘀嗒后第一次触发 Task1，每 300 时钟嘀嗒触发 Task1 一次。创建这些任务的代码如下。

```
void Task1(void)
{
    Uart_Printf("Task1 schedul by timer\n");
}
g_hTaskHandle_1 = AddTask(Task1 ,   0x60 , 64 );
TX_TIMER Timer1;
_txe_timer_create(&Timer1, " Timer1",    Task1, Null, 2000, 3000, 1);
```

运行 KVM 系统，在超级终端上可以看到 Task1 周期性的被 Timer1 触发而运行。如图 5.3 所示。可以说明 KVM 系统定时器任务的定时功能正确。

在正确性得到验证的情况下测试定时器的精确性。采用五组测试集。五组分别含定时器任务的数量为 101，201，301，401，501。所有的定时器设置为单次定时，即仅仅定时一次。

设某个测试集的定时器任务数量为 N。该测试集中第一个定时器任务的作用是标识起始测试点。它触发的普通任务是在超级终端上打印出“开始测试”的字样。它的到期时间设置的较大，为 10000 时钟嘀嗒，这样是为了减小测试人员看见“开始测试”字样时记录测试起点时的时间误差。第二个定时器任务的到期时间设置为

11000 时钟嘀嗒，时间到期时触发普通任务 Task2,第三个定时器任务的到期时间设置为 12000 时钟嘀嗒，时间到期时触发 Task3,依次类推第 N 个定时器任务的到期时间设置为 10000+(N-1)*1000 时钟嘀嗒,时间到期时触发 TaskN。当 TaskN 被触发时，记录结束时间。

设测试结束时间与起始时间的间隔为 T。KVM 系统一个时钟嘀嗒的时间为 t。理论上定时器经历了(N-1)*1000 个时钟嘀嗒，设该数值为 T1。但实际经历的时钟嘀嗒数为(T/t)，设该数值为 T2。

表达式(T2-T1)/T1 反映了 KVM 系统的时延性，也就是定时器的精确性。每个测试集测量三次。第一次设置系统时钟嘀嗒的间隔为 0.5 毫秒，第二次为 1 毫秒，第三次为 1.5 毫秒。测量数据如表 5.2 所示。

（1）测试结果及分析

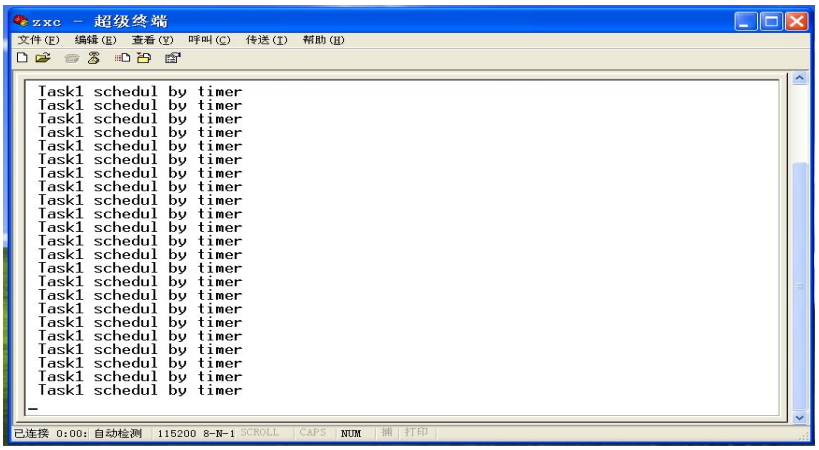

图 5.3　普通任务 Task1 被定时器任务 Timer1 周期性触发

由图 5.3 可知 KVM 系统定时器功能正确。

表 5.2　各测试集定时器平均时延

序号	定时器任务数量	时钟嘀嗒间隔(ms)	时延(时钟嘀嗒倍数)
1	101	0.5	0.147
2	101	1	0.138
3	101	1.5	0.124
4	201	0.5	0.149
5	201	1	0.140

续表 5.2　各测试集定时器平均时延

序号	定时器任务数量	时钟嘀嗒间隔(ms)	时延(时钟嘀嗒倍数)
6	201	1.5	0.127
7	301	0.5	0.150
8	301	1	0.142
9	301	1.5	0.129
10	401	0.5	0.152
11	401	1	0.145
12	401	1.5	0.132
13	501	0.5	0.154
14	501	1	0.146
15	501	1.5	0.133

根据表 5.3 所记录的定时器平均时延迟数据，定时器平均时延迟与定时器数量的关系如图 5.4 所示。

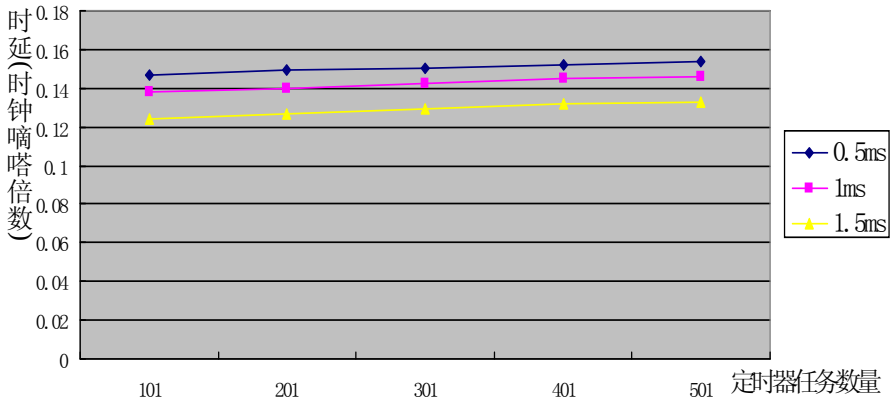

图 5.4　定时器平均时延与定时器任务数量对比图

由图 5.4 可知，随着 KVM 系统定时器数量的增多，定时器的平均时延呈递增趋势。同时时钟嘀嗒的间隔时间越小，定时器由于时延产生的误差效果越明显。分析图 5.4 的时延数据，可知 KVM 系统定时器的平均时延数据保守为 0.14 倍时钟嘀嗒间隔时间，这在误差范围内对于实时操作系统是可以接受的。

5.2.3 任务调度性能测试分析

KVM 系统调度模块根据任务的优先级进行调度。对测试任务设置优先级，然后经过调度模块调度。在一定的数量范围内记录相关优先级任务被调度运行的次数，以此来对 KVM 系统调度模块的调度性能做出分析。

（1）任务调度测试案例的设计

传统的 uC/OS-II 不支持优先级相同的任务间轮转调度。现测试 KVM 系统对轮转调度的支持，创建 10 个普通任务，它们的优先级被设置为相同。为了测试方便每个任务执行的是打印出一条标识自身的语句。运行系统，测试结果如图 5.5 所示。10 个任务被轮转调度。

针对不同优先级的任务调度情况采用测试集进行测试。分为五组测试集，第一组含 30 个任务。设 30 个任务的优先级顺次排列从优先级 1 一直到优先级 30。第二组包含 60 个任务，30 个优先级每个优先级对应两个任务。第三组包含 90 个任务，30 个优先级每个优先级对应三个任务。依次类推，第五组包含 150 个任务，30 个优先级每个优先级对应 5 个任务。

对每组测试集进行调度测试。记录以下三类任务被调度的次数。第一类：优先级处于 1-10，即优先级较高的任务。第二类：优先级处于 11-20,优先级居中的任务。第三类：优先级处于 21-30,优先级较低的任务。每组测试后，三类任务的调度次数情况如表 5.3 所示。

（2）测试结果及分析

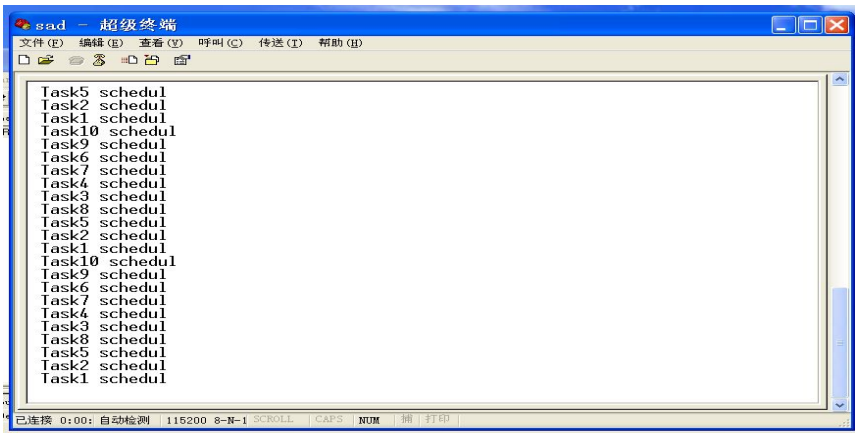

图 5.5 10 个优先级相同的普通任务轮转调度

表 5.3　三类任务调度次数

测试集	优先级在 1-10 范围内的任务调度次数	优先级在 11-20 范围内的任务调度次数	优先级在 21-30 范围内的任务调度次数
30	529	301	170
60	475	342	183
90	443	352	205
120	430	358	212
150	422	362	216

由表 5.3 可知，KVM 系统调度模块保证了高优先级的任务有较高次数的调度。

5.3 本章小结

本章主要是对 KVM 系统任务管理机制进行测试。根据测试数据分析得出 KVM 系统任务管理机制在性能上能够达到一定的实时性，可靠性。

6 全文总结和研究展望

本文在分析国内外相关工作的基础上，对当今应用较为广泛的专用操作系统的任务管理机制进行了分析和研究，目的是设计并实现一个专用操作系统 KVM OS 中的任务管理机制。本章将对所做的工作进行总结并对未来工作进行展望。

6.1 全文总结

本文的特点是理论研究与工程设计结合。理论研究工作包括对嵌入式平台 ARM9 处理器结构的分析、定时器工作原理的分析、操作系统常见调度算法的分析；工程设计主要围绕如何为专用操作系统 KVM OS 设计并实现高效的任务管理机制展开，包括 KVM 系统任务调度机制、任务切换机制、定时器机制的设计，任务控制块的设计，API 函数的设计等。

本文所做的工作主要有以下几个方面：

（1）对 KVM 系统的运行平台 ARM9 处理器进行了分析和研究，给出了 KVM 系统中的任务切换机制的关键技术和架构设计。

（2）设计出了一种新的定时器机制的原理和模型。这种定时机制在数学建模和算法流程上都给出了支持大规模数量的定时器任务并行定时的研究思路。

（3）在研究常见操作系统调度算法的基础上，设计出了一种时间复杂度低同时满足公平性的调度算法。

（4）在理论研究的基础上，实现了多任务管理机制中的任务切换模块、定时器管理模块、任务调度模块，同时实现了提供给应用层调用的任务管理 API 函数。

6.2 研究展望

为了进一步提高 KVM 系统任务管理的性能，下面几个方面待进一步的研究：

（1）中断机制的改进

当今很多实时操作系统都支持嵌套中断。目前 KVM 系统仅支持单级中断，暂不

支持嵌套中断。因此应该设计一套嵌套中断机制，以加强任务中断的处理机制。

（2）逻辑地址的引入

KVM 系统任务管理模块中涉及到的内存地址均是实际物理地址，以后当系统内存扩展，任务数量增多后，仍然访问实际物理地址显然是不合适的。因此应该引入逻辑地址的概念同时设计一种逻辑地址与物理地址映射的管理机制，以提高多任务条件下的访问内存的性能。

（3）定时器任务与普通任务统一化的改进

KVM 系统根据功能将任务划分为普通任务和定时器任务两个类别。定时器任务的优先级高于普通任务的优先级。两种不同任务的管理模块差别较大，这对以后系统的功能扩展是不利的。所以在以后的研究中应该将定时器任务和普通任务逐渐融合为一种类型的任务。

（4）辅助模块的完善

缺乏相应的 GUI，给 KVM 系统任务管理的性能测试带来一定困难。另外随着当今操作系统对 TCP/IP 协议的支持，网络化的发展。KVM 系统任务管理机制应该添加相应的网络处理模块。

致 谢

首先感谢我的导师林安副教授，他对学术严谨的态度，扎实的理论基础都是我做人做学问的榜样。从论文的选题到论文的书写林老师都给与了我详细的指导。在学业即将完成之际，向林老师致以最诚挚的敬意和最衷心的谢意。

我还要感谢课题组谢长生教授、黄浩副教授、胡迪青副教授、王海卫副教授、万继光老师、吴非老师、肖亮老师对我的关心和帮助。还有许多给过我帮助的其他老师，在此也表示深深的谢意。我还要感谢课题组的其他成员，博士生叶磊，硕士生蔡其威、王翔、余胜永以及所有给予我帮助的同学和朋友们，我永远也不会忘记和他们一起度过的时光，这将是我的人生一段值得永远怀念的时光。

最后，我要深深的感谢我的父母。他们在我的背后给予了我无尽的帮助。最后借此机会向所有关心、支持、帮助过我的老师、同学表示由衷的感谢！

参考文献

[1] Sinnott. R. O, Stell. A. J, Watt. J. Experiences in teaching grid computing to advanced level students. CCGrid 2005. IEEE International Symposium. May 2005: 51~52

[2] 王海燕. uC/OS-II中任务调度的改进与实现. 现代电子技术, 2006, 14(2): 41~42

[3] 徐晓磊, 董兆华, 吴建峰等. Linux可抢占内核的分析. 计算机工程, 2003, 29 (15): 3~5

[4] Terrasa Andres, Ana Garcia-Fornes, Vicente J. Botti. Flexible Real-Time Linux: A Flexible Hard Real Time Environment. Real-Time System. 2002, 22(2): 151~173

[5] 张忠, 樊留群. VxWorks在S3C2410上的BSP设计. 微型电脑应用, 2005, 21(10): 21~23

[6] 冯习宾, 蒋廷彪, 刘慧. 基于ARM的嵌入式数控系统的研究. 世界科技研究与发展, 2008, 10(2): 589~591

[7] 傅明. 嵌入式操作系统通信和同步机制的研究: [硕士学位论文]. 湖南: 长沙理工大学, 2005

[8] B. Srinivasan. A firm real-time system implementation using commercial off-the-shelf hardware and free software, in Proc. IEEE Real-Time Technology and Application Symposium, June 1998. 112~19

[9] Yanbing li, Miodrag Potkonak, Wayne Wolf. Real-Time Operating System for Embedded Computing, May 2001, 21~22

[10] E. Akhmetshina. A Tiny Operating System for Extremely Small Embedded Platforms. International Conference on Embedded Systems and Applications. 1996: 116~122

[11] Moses J. PSO. Select the smallest modular real-time operating system for embedded applications. REAL-TIME Magzine, 1995(2): 66~68

[12] 王晓鸣, 王树新, 张宏伟. 实时操作系统uC/OS-II在ARM上的移植. 机电一体化,

2007, 1(2): 56~57

[13] 高富强, 秦昌硕, 游纪原等. uC/OS-II内核扩充时间片轮转调度算法的设计. 计算机应用, 2009, 4(3): 1128~1130

[14] 王劲松, 李正熙, 夏旺盛. 嵌入式操作系统uC/OS-II的内核实现. 现代电子技术, 2003, 8(2): 48~49

[15] 那加. 嵌入式实时操作系统uC/OS-II在SharpLH79520处理器上的移植. 测控技术, 2007, 10(4): 53~54

[16] 朱显新, 黄涛, 卢珞先. uC/OS-II和uClinux的比较. 单片机与嵌入式系统应用, 2004, 10(3): 5~7

[17] 王志坚, 梁军龙. uC/OS-II在角位置传感器中的应用. 山西焦煤科技, 2008, 12(5): 3~4

[18] 魏迎梅, 王涌. Linux使用指南(第二版). 北京: 电子工业出版社, 2002. 43~45

[19] 毛得操, 胡希明. Linux内核源代码情景分析(第一版). 浙江: 浙江大学出版社, 2001. 615~620

[20] 赵炯. Linux0. 11内核完全注释(第一版). 北京: 机械工业出版社, 2004. 55~57

[21] Jean J. Labrosse. 嵌入式实时操作系统uC/OS-II(第二版). 邵贝贝译. 北京: 航空航天大学出版社, 2003. 134~137

[22] Luca Abeni, Ashvin Goel, Charls Krasic, et al. A Measurement-Based Analysis of the Real-Time Performance of Linux. Proceeding of the Eighth IEEE Real Time and Embedded technology and Applications Symposium (RTAS'02), IEEE Computer Society Washington, DC, USA, May 2002, (1): 133~142

[23] A. Goel, L. Abeni, C. Krasic, et al. Supporting time-sensitive applications on a commodity OIn 5th Symp. May 2002: 165~180

[24] Y. Wang, K. Lin. Implementing a General Real-Time Scheduling Framework in the RED-Linux Real-Time Kernel, in Proc. of RTSS'99, phoenix, Arizona, May 1999. 246~255

[25] J. P. Lehoczky, L Sha, J. K. Stronsnider. Enhanced Aperiodic Responsiveness in

Hard Real Time Environments. In: Proc of the 8th IEEE Real Time System Symposium. San Jose, California: IEEE Computer Society Press, May 2004: 210~217

[26] 吕跃刚, 张新房, 徐大平等. Windows CE在嵌入式工业控制系统中的应用思考. 单片机与嵌入式系统应用, 2002, 9(6): 11~12

[27] 王旭东, 徐刚. 基于Windows CE4. 2嵌入式操作系统短信通讯的应用研究. 全国第17届计算机科学与技术应用(CACIS)学术会议论文集(下册), 2006. 916~918

[28] 缪昌国, 贾民平, 胡建中. 基于Windows CE. net的嵌入式状态监测软件研发. 机械制造与自动化, 2009, 38(01): 125~126

[29] 马德荣, 桑楠, 莫彩文. VxWorks下串行通信的实现原理与应用. 实验科学与技术, 2005, 2(11): 48~49

[30] 桂肖敏, 罗飞, 曹建忠. 嵌入式操作系统VxWorks在ARM芯片上的应用. 嵌入式操作系统应用, 2006, 22(32): 101~102

[31] 许志民, 李翔, 李苏桥. 一种基于VxWorks的高实时性软件架构设计. 电讯技术, 2007, 47(2): 199~202

[32] 董晓霞. 基于QNX实时操作系统的编程应用. 现代电子, 2000, 3(7): 34~35

[33] 代科学, 李强, 母其勇. 基于QNX的网络视频监控系统的设计与实现. 计算机测量与控制, 2003. 11(3): 189~190

[34] 危淑敏, 苗克坚. 基于QNX的实时嵌入式测试系统设设计. 合肥工业大学学报(自然科学版), 2007. 30(11): 1408~1409

[35] 赵艳明, 全子一. pSOS在窄带ISDN可视电话软件中的应用. 数据通信, 2002, 2(4): 31~32

[36] 刘峰, 朱秀昌. 嵌入式实时操作系统pSOS在多媒体通信终端中的应用. 电视技术, 2001, 4(8): 24~25

[37] Bill O. Gallmeister, Chris Lanier. Early Experience With POSIX 1003. 4 and POSIX 1003. 4A. Proceeding of the IEEE Symposium on Real-Time Systems. June 1991: 190~198

[38] Balarin, F. Lavagno, L. Murthy. Scheduling for Embedded Real-Time Systems Design &Test of Computers, IEEE, 1998, 15(1): 71~74

[39] M. Aron, P. Druschel. Soft timers: efficient software timer support for network processing. ACM Trans. Comput. Syst. June 2000, 18(3): 197~228

[40] John Coodacre, Andrew N. Sloss, Parallelism and the ARM Instruction Set Architecture, Computer Archive, July 2005, 38. 232~237

[41] 杜春雷. ARM体系结构与编程(第二版). 北京: 清华大学出版社, 2003. 250~253