一种微内核任务调度的实现方法

陈 杰
(杭州电子科技大学 智能与软件技术研究所,浙江 杭州 310018)

摘要:为解决微内核技术在小内存 CPU 芯片中应用受限问题,基于双层裁剪技术,将系统任务裁剪分类,并开辟相应优先级线程进行调度管理,提出了一种利用小线程实现微内核任务调度的方法,实现了微内核技术在模糊电饭煲上的实验应用。试验结果表明:该方法实现简单,系统开销小,可移植性强,稳定性高,可应用于小 CPU 上的嵌入式系统。

关键词:微内核;线程调度;任务调度;任务裁剪
中图分类号:TP316.2 **文献标识码**:A **文章编号**:1001 - 4551(2010)05 - 0078 - 04

Approach for realization of micro-kernel task scheduling

CHEN Jie
(Institute of Software and Intelligent Technology, Hangzhou Dianzi University, Hangzhou 310018, China)

Abstract: Aiming at realizing micro-kernel technology limitedly used in micro-CPU, classifying and cutting out system tasks based on double-cut technology, managing them by opening corresponding priority dispatching threads, an approach for using small threads was given to realize micro-kernel task scheduling, and the application of micro-kernel technology in fuzzy rice cooker was realized. The research results indicate that the method is simple, small system overhead and portability, high stability, and it can be applied to pocket embedded systems based on micro-CPU.

Key words: micro-kernel; thread scheduling; task scheduling; task disassembling

0 引 言

微内核是一个小型操作系统的核心,只有最基本的操作系统功能才放在内核中,非基本的服务和应用程序在微内核之上构造,并在用户模式下执行[1]。因其内核精巧、结构模块化强、系统可靠性高等优点,微内核已成为当前操作系统的发展方向和趋势。Mach[2],Amoeba[3],Chorus[4],Qnx[5]等是较成功的微内核操作系统,并已得到广泛使用。

不可否认,微内核技术已经取得了很大成功,但是人们在研究和使用微内核的过程中,发现其普遍存在一个严重问题—性能不好、效率不高[6]。微内核结构直接决定其性能[7],μC/OS Ⅱ等大部分微内核的结构都较为复杂、臃肿,致使其内存开销过大、稳定性不高,在面对小内存 CPU 芯片上的嵌入式系统时显得无能为力,尤其是 FLASH 仅有几千字节时,根本无法满足系统应用要求。对各类嵌入式系统任务进行分析研究,可将其归为 3 类:实时任务、快速任务和慢速任务。

为了将微内核应用于小内存 CPU 芯片上的嵌入式系统,本研究提出一种利用小线程实现微内核任务调度的方法,将内核结构简单化,以减少其内存开销,提高系统性能。

1 任务调度

1.1 任务模型

将系统任务分为周期性任务和非周期性任务,本研究约定:

(1) $R = \{R_1, R_2, \cdots, R_n\}$ 是含有 n 个周期任务的集合。

任务以 $R_i = \{T_i, C_i, D_i\}$ 表示,其中,T_i, C_i, D_i 分别表示任务 R_i 的周期、执行时间和截止期。

收稿日期:2009 - 09 - 30
作者简介:陈 杰(1984 -),男,福建泉州人,主要从事嵌入式技术、智能与软件技术方面的研究. E-mail: cj_841010@163.com

(2) $R' = \{R_{n+1}, R_{n+2}, \cdots, R_{n+m}\}$ 是 m 个非周期性任务的集合。

任务以 $R_j = \{C_j, D_j, K_j\}$ 表示,其中,C_j, D_j, K_j 分别表示任务 R_j 的执行时间、截止时间、关键度。

本研究假定:

(1) 优先级高的任务可以抢占优先级低的任务;

(2) 任务不被挂起;

(3) 任务切换时的开销忽略不计;

(4) 相同优先级任务按先来先服务的顺序处理;

(5) 每个任务新一轮的请求均在这个任务结束之后;

(6) 任务间相互独立(不存在因共享资源而导致的阻塞)。

1.2 任务裁剪

1.2.1 设计思想

基于上述任务模型,本研究采用双层裁剪算法对用户任务进行裁剪分类,最终将用户任务转为内核任务。第 1 层裁剪算法将用户任务裁剪成尽可能少的可调度的几个优先级任务集合,以减少任务调度的系统开销;第 2 层裁剪算法则是针对第一层任务剪裁结果,再裁剪成几个不同优先级等级的任务集合,以便让用户任务在一个更可靠、更合理、更稳定的环境下运行。

1.2.2 裁剪算法

第 1 层裁剪算法采用邢建生等人提出的静态最少优先级分配算法(简称 LNPA 算法[8-9]),目的是为了获得最少优先级的任务集,以减少线程调度开销。

第 2 层裁剪算法采用宾雪莲等人提出的基于有限优先级的静态优先级分配算法(简称 AGP 算法[10]),目的是为了获得最优最合理的任务集,以协调各个任务高效执行。

采用静态有限优先级分配的主要目的是在任务集合静态优先级可调度的情况下,以较小或最小的代价(较少或最少的优先级)保持任务集合可调度,从而最大化系统的性价比。而双层算法之所以不采用同一种算法,是因为第 1 层裁剪是为了获得最少的优先级,而第 2 层裁剪是为了获得最优的任务集。

1.2.3 任务裁剪应用

假设共有 n 个任务,任务 i 以 Task(i) 表示。采用第 1 层裁剪算法,将用户任务裁剪成 m 个优先级的内核任务,即{TASK(1),TASK(2),…,TASK(m)};针对第 1 层裁剪结果,再通过第 2 层裁剪算法,各个内核任务再被裁剪成 L 个优先级的子任务集,即 TASK(i) = {TASK(i,1), TASK(i,2), …, TASK(i,L)},其中,TASK(i,j) = {Task(a),…,Task(b)},任务集 TASK(i,j)中有 1 个或多个任务。任务裁剪过程如图 1 所示。

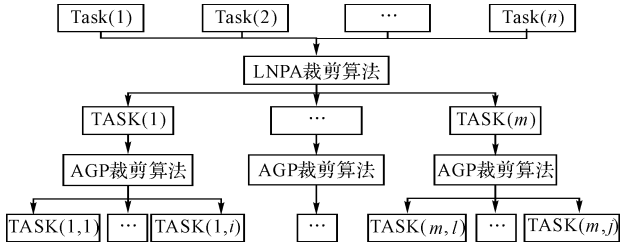

图 1　任务裁剪流程图

1.3 调度策略

1.3.1 设计思想

本研究采用双层调度策略,第 1 层为线程调度,第 2 层为任务调度。针对第 1 层裁剪算法所得裁剪结果的 m 个优先级内核任务,开辟相应 m 个优先级线程,并采用静态优先级抢占式调度算法进行管理。每个线程内包括 1 个或多个经过第 2 层裁剪算法所生成的任务(优先级可以相同),其中,针对不同优先级任务,采用静态优先级非抢占式调度算法;针对相同优先级的任务,采用先来先服务调度算法。双层调度结构图如图 2 所示。

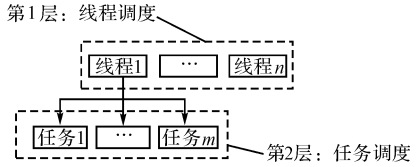

图 2　双层调度结构图

1.3.2 算法实现

假设共有 m 个线程,Thread(i)表示线程 i,线程优先级由高到低依次为 Thread(0), Thread(1), …, Thread(m-1),优先级越高,优先权越高。

(1) 线程就绪任务链表结构体定义如下:

```
typedef struct NormalList
{
    unsigned int TaskAddr;//任务入口
    unsigned char Priority;//任务优先级
    struct NormalList * next;
}List, ListNode;
```

(2) 线程参数:

```
Run[m];//执行标志
Ready[m];//就绪标志
PauseSP[m];//阻塞现场堆栈
ThreadTasks[m];//线程内就绪任务链表
```

(3) 事件处理:

每当有新任务到达时,便将任务所属线程的就绪标志置 1,并把此任务按优先级高低顺序插入到该线程的就绪任务链表中。

(4) 线程调度的触发方式:

线程调度触发由时钟中断产生,中断处理流程如图 3 所示。

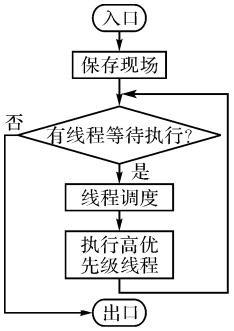

图 3 中断处理流程图

定义 1:线程阻塞是指正在执行的线程被时钟中断打断而被中止的现象。

定义 2:保存现场是指线程阻塞时现场寄存器压入堆栈后的堆栈指针地址存于相应线程的阻塞变量过程。

(5) 第 1 层线程调度算法(伪代码):

```
void ThreadScheduling(void)
{
    n = 0; //从高优先权线程开始判断
    while(n < m)
    {
        if (Run[n] = = 1) //线程 n 未执行完
              恢复线程 n 的阻塞现场并执行;
        else if (Ready[n] =1)//线程 n 就绪
        {
            执行线程 n;
            重新开始线程调度;
        }
        else
            n + + ;//下一线程
    }
}
```

(6) 第 2 层任务调度算法(伪代码):

```
void ThreadFun(unsigned char i)
{
    Run[i] = 1;//线程正在执行;
    while(线程 i 内有就绪任务)//任务调度
    {
        取出线程 i 任务链表首结点任务;
        删除此结点;
        执行此任务;
    }
    Ready[i] = 0;//清就绪标志
    Run[i] = 0;//线程结束
}
```

1.3.3 算法分析

任务包括就绪、执行、阻塞 3 种状态,常规调度算法是在这 3 种状态间进行切换以实现任务调度,而本调度策略摒弃 3 种状态的切换,降低了调度开销。第 1 层调度策略采用静态优先级抢占式线程调度,及时地响应实时任务和快速任务的请求,保证系统实时性;第 2 层调度策略采用混合调度算法,任务响应快,及时地协调各个任务的执行,并且任务调度及切换的开销甚小,使系统的稳定性、可靠性和实时性都得到加强。

2 实验应用

2.1 实验描述

本研究采用新技术 Cortex-M3 内核 LM3S101 (Flash:8 KB,SRAM:2 KB)作为硬件平台,实现模糊电饭煲的开发应用。根据模糊电饭煲的工作原理,系统任务主要包括:温度采集、温度控制、开关量控制、数据显示、时间控制、故障检测控制和报警控制。

笔者采用第 1 层裁剪算法将系统任务裁剪成以下 3 类内核任务:

任务 1:温度采集、开关量控制、报警控制;

任务 2:温度控制、数据显示、时间控制;

任务 3:故障检测控制。

针对这 3 类内核任务,本研究开辟了 3 个线程,采用静态优先级抢占式调度算法进行管理。线程 1 用于管理任务 1,其优先级最高;其次为线程 2(用于管理任务 2);线程 3(用于管理任务 3)的优先级最低。优先级高的线程可以抢占优先级低的线程。

采用第 2 层裁剪算法将内核任务 1 裁剪成 3 个子任务,即温度采集、开关量控制和报警控制,其中,温度采集的优先级最高,其次为开关量控制,报警控制的优先级最低,这 3 个任务采用静态优先级非抢占式调度算法进行管理。同样,采用第 2 层裁剪算法将内核任务 2 裁剪成 3 个子任务,优先级由高到低为温度控制、数据显示和计时控制,采用静态优先级非抢占式调度算法进行管理。

在系统实现时,各个内核任务均要开辟自己独立的数据空间,尤其是存在数据共享的任务,以避免优先级高的任务抢占时破坏了优先级低的任务的数据区。如温度控制正在进行控制时,被温度采集抢占了,如果它们共用数据区,此时温度采集就会改变数据区,造成温度控制的混乱,因此这两个任务必须要有各自的数据区,并规定:在温度控制任务去读温度采集数据的时候,温度采集任务不能抢占温度控制任务(可通过禁止中断实现),只有在数据读取完后才可抢占,温度控制期间也可以抢占,这样就可以保证温度控制数据区在本次控制内数据不变。

2.2 实验结果

(1) 内存开销:时钟中断现场保护内存开销为 24 个字节(6 条汇编指令),内核的线程调度及线程切换内存总开销为 164 个字节(41 条汇编指令),线程内的任务调度内存开销为 72 个字节(18 条汇编指令)。微内核任务调度内存开销不会随着系统任务数的变化而变化。

(2) 时间开销:假设系统平均执行一条汇编指令的时间为 T,则线程调度及线程切换的时间开销如表 1 所示,其中线程切换的时间开销基本上是固定的,但是线程调度的时间开销会随着线程数量的增多而增大。

线程抢占时间开销会随着线程数量的增多而增大,但由于内核的线程数量已优化,线程抢占时间开销不大,能满足实时性的要求。

线程抢占时间 = 保存现场时间 + 线程调度时间 + 线程切换时间

表 1　时间开销

时间开销	最好时	最坏时	平均
线程调度	$6T$	$72T$	$39T$
线程切换	$8T$	$11T$	$9.5T$
线程抢占时间	$25T$	$35T$	$30T$

3　结束语

本研究针对微内核在小内存 CPU 芯片中应用受限问题进行了分析研究,提供了一种利用小线程实现微内核任务调度的方法,并将其思想应用到实践中,得出了更优的微内核实现方法。

研究结果表明,该微内核任务调度内存开销小,可适用于小内存 CPU 芯片上的嵌入式系统;内核结构简单,增强了系统稳定性和可靠性;调度量小,提高了任务响应率,保证了系统实时性。

参考文献(References):

[1] 潘　清,张晓清. 操作系统微内核技术研究[J]. 软件学报,1998,9(8):609 - 612

[2] ACCETTA M, BARON R, BOLOSKY W, et al. Mach: a New Kernel Foundation for Unix Development[C]//Proc. of the USENix Summer 86 Conference. Atlanta, GA: [s. n.], 1986:93 - 113.

[3] MULLENDER S J, ROSSUM G V, TANENBAUM A S, et al. Amoeba—a distributed operating system for the 1990s [J]. **IEEE Computer Magazine**,1990,23(5):44 - 53.

[4] ROZIER M, ABROSSIMOV A, ARMAND F, et al. Overview of the Chorus Distributed Operating System[C]//Proc. of the USENIX Workshop on Micro-Kernels and Other Kernel Architectures. Seattle: [s. n.],1992:39 - 69.

[5] HILDEBRAND D. An architectural overview of QNX[C]// Proc. of the USENIX Workshop on Micro-kernels and Other Kernel Architectures. Seattle: [s. n.],1992:113 - 126.

[6] LIEDTKE J. Toward real microkernels[J]. **Communications of the ACM**,1996,39(9):70 - 74.

[7] 王世铀,郭福顺,臧天仪. 微核心操作系统的结构对性能的影响[J]. 计算机研究与发展,1999,36(1):57 - 61

[8] 刘福岩,尤晋元,曾国荪. 通过单地址空间提高微内核操作系统的效率[J]. 计算机工程,2000,26(9):48 - 50.

[9] 邢建生,王永吉. 一种静态最少优先级分配算法[J]. 软件学报,2007,18(7):1844 - 1854.

[10] 宾雪莲,杨玉海,金士尧. 一种有限优先级的静态优先级分配算法[J]. 软件学报,2004,15(6):815 - 822.

[编辑:李　辉]