

RTAI 实时调度算法及其调度器的改进

李学桥, 梁 爽, 陈 园

(郑州轻工业学院 计算机与通讯工程学院, 河南 郑州 450002)

摘 要: 在实时系统中, 任务调度策略是内核设计的关键部分. 如何进行实时的任务调度, 使任务能在特定的周期内完成是实时操作系统领域研究的一个热点问题. 文中将一种基于 RM 算法的改进算法 CPSS 算法引入到 RTAI 调度器中, 针对 RTAI 调度器在系统过载情况下出现调度性能下降等缺点, 对 RTAI 调度器进行优化和改进. 对改进后的调度器在调度时延方面和调度算法仿真方面进行了测试, 实验证明了改进后的调度器能够提高 Linux 系统的实时性.

关键词: 实时系统; 任务调度; RTAI; 调度器; CPSS 算法

中图分类号: TP316.2

文献标识码: A

文章编号: 1000-7180(2011)04-0147-04

The Improvement of RTAI Real-Time Scheduling Algorithm and Scheduler

LI Xue-qiao, LIANG Shuang, CHEN Yuan

(College of Computer and Com. Eng., Zhengzhou University of Light Ind., Zhengzhou 450002, China)

Abstract: In real-time system, task scheduling policy is the key part of kernel design. How to design task scheduling to ensure that all the tasks will be completed before its deadline is an important problem in the research on real-time operating system. Because the scheduler of RTAI has a bad performance when the system is in heavy load or overload, this article adds a Comprehensive Priority Static Schedule (CPSS) algorithm to RTAI scheduler, and improves and optimizes the RTAI scheduler. Finally, a simulator of the scheduling algorithms and a test of the scheduler are presented. The experiment has proved that improved scheduler can increase the real time Linux system.

Key words: real-time system; task scheduling; RTAI; scheduler; CPSS algorithm

1 引言

随着计算机技术和电子技术的快速发展, 嵌入式产业也随之迅速崛起, 成为了当今社会人们最关注的产业之一. 而在嵌入式产业中, 目前比较成熟的嵌入式操作系统有 VxWorks, PSOS 和 Windows CE^[1]等, 而 Linux 操作系统以其源代码开放, 内核的可移植性, 高效性, 健壮性, 高效的网络通信支持能力和丰富的开放应用软件等优秀, 已经成为了最主流的操作系统之一, 上述优点也使得 Linux 系统在嵌入式领域得到了快速发展. 但 Linux 系统是典型的分时操作系统, 所以必须对 Linux 系统进行改进以更好的适用嵌入式实施应用的需要. 目前有两

种对 Linux 系统进行实时化改造的方案: 对内核直接修改和双内核^[2]. 而修改内核只能达到软实时的目的, 因此文中使用双内核方案对 Linux 系统进行实时化改造.

2 RTAI 方案及其实时调度策略

RTAI(Real-Time Application Interface)是对 Linux 内核的硬实时扩展, 它遵循自由软件规范; 它可以提供工业级的 RTOS 功能, 而且其所有的功能都可无缝的通过 GNU/Linux 环境访问. RTAI 项目是由意大利米兰理工学院航天工程系(DIAPM)发起开发的遵循 GPL 的开源项目. 目前 RTAI 已经支持 x86、PowerPC、ARM、MIPS、CRIS 等处理器,

是目前支持处理器最多的 Linux 硬实时解决方案之一. 已经有很多工业控制系统采用 RTAI 作为运行支撑环境, 它也表现出可以与商业实时操作系统相媲美的性能^[3].

RTAI 还并不能算是一个完整的操作系统, 而只是一个具备操作系统核心的实时内核. 在双内核系统中, RTAI 代替 Linux 接管了所有的硬件资源. RTAI 调度器把 Linux 系统内核作为一个最低优先级的任务, 只有当实时任务都运行完之后才能运行.

RTAI 调度器主要采用的是 RM 调度算法和 EDF 调度算法^[4-5]. RM 调度算法是一种静态的调度算法, 算法简单, 实现容易, 但 CPU 利用率不高. EDF 算法是一种动态的调度算法, CPU 利用率高, 但经常检查不出即将错过截止期的任务, 导致大量任务错过其截止期, 引起系统性能下降. 如今实时任务在嵌入式领域应用的越来越广泛, 因此对实时任务调度器提出了很高的要求^[6].

3 基于 RM 算法的改进调度算法—CPSS

RM 调度算法是一种非常经典的静态周期任务调度算法. 该算法简单, 容易实现但并不容易实现最紧急的任务. 文中介绍了一种基于综合优先级的静态调度算法 (comprehensive-priority static schedule, CPSS), CPSS 算法是基于“综合优先级”这个参数, 这里的综合优先级是结合了任务的运行时间 E_i 和重要程度 I_i 两个特征参数的综合优先级.

记任务 τ_i 的重要程度为 I_i , 再记任务 τ_i 的运行时间为 E_i , 定义任务 τ_i 的综合优先级为 P_i , 则:

$$P_i = P(\tau_i, I_i, E_i) = W_i I_i + W_e E_i$$

其中, W_i, W_e 分别为重要程度和运行时间的权值, 且 $W_i > W_e$, 其中 $W_i + W_e = 1$, P_i 越大, 优先级越高.

综上, 系统的综合优先级是通过结合算法中的两个参数 (重要程度 I_i 和周期 E_i) 计算得出的值, 而不仅仅是取决于算法的周期.

算法中, 将队列按综合优先级的排序分为两个队列, 关键任务集 S 和非关键任务集 N , 优先对关键任务集 S 的任务进行调度. 首先判断 S 队列中是否还有任务, 如果有则按照 FIFO 算法继续进行下一轮的调度, 如果没有则将 N 队列中的综合优先级最高的任务放入 S 队列中继续执行. 执行直至完成. 本算法优先对 S 队列中的任务进行调度. 只有当 S 队列中没有可以运行的任务时才对 N 队列进行调度, 据此可以保证系统优先调度关键任务集 S 中的

任务.

改进的 CPSS 调度算法既延续了 RM 调度算法实现简单等优点, 又改进了 RM 算法经常忽略重要任务的缺点. 使一些周期长但重要程度很高的任务能尽快得到调度, 改进了 RM 算法.

4 针对 CPSS 改进算法的 RTAI 调度器的改进

针对 CPSS 算法对 RM 算法的改进, RTAI 调度器需要从数据结构, 任务队列管理和定时器设置等三部分进行改进.

4.1 对数据结构的改进

任务控制块是 RTAI 中最重要的一个数据结构体. RTAI 与所有的进程管理程序一样, 对于每一个实时任务都有一个任务控制块. 任务控制块就像进程的“户口”一样, 记录着进程的各种属性和进程占用的各项资源^[6]. 在 `rtai_sched.c` 中以结构体 `rt_task_struct` 表示任务的控制块.

改进前的 RTAI 调度器任务控制块的主要数据结构有: `base_priority` 表示任务的优先级; `period` 表示任务的周期; `policy` 表示的是调度策略, 其中大于 0 表示使用 RM 算法, 小于 0 表示使用 EDF 算法, 改进的 CPSS 算法对 `policy` 不好再扩展, 因此需要再引进一个数据结构体表示是否使用 CPSS 算法. 具体程序段如下:

```
typedef struct rt_task_struct {
    ...
    /* Append for CPSS algorithm */
    int CPSS_policy; // 程序是否使用 CPSS 算法
    int CPSS_importance; // 任务的重要程度; 值越大任务越重要
    int CPSS_compre_priority; // 任务的综合优先级
    int Wi, We; // 分别为重要程度和周期的权值
    RTIME CPSS_sched_deadline; // 实时任务调度的相对截止期
    RTIME CPSS_exec_time; // 执行完当前任务需要的时间
    RTIME CPSS_current_exec_time; // 任务的当前执行时间, 随任务执行而变化
    /* Append for CPSS algorithm end */
    ...
}
```

4.2 对任务队列管理的改进

RTAI 内核中本身存在着三种任务调度队列, 分别为任务队列, 任务延时队列和任务就绪队列. 在

RTAI 内核中本身存在着一些函数,它们分别为:

```
enq_ready_task(); //将任务按照 RM 算法插入就绪队列
enq_ready_edf_task(); //将任务按照 EDF 算法插入就绪队列
```

文中引入 CPSS 算法,如果在原有的任务队列管理中新加入 CPSS 算法插入就绪队列,那样会更改很多原有操作. 文中将 CPSS 算法与 RM 算法相结合,将 RM 算法本身比较任务的优先级语句改为比较 CPSS 算法中通过计算出来的综合优先级,以此来确定是否进行调度. 通过在 `enq_ready_task()` 中增加比较综合优先级的语句来进行改进. 具体扩展后的代码如下:

```
static inline void enq_ready_task(RT_TASK * ready_task)
{
    RT_TASK * task;
    ...
    /* modify for CPSS algorithm */
    CPSS_compre_priority = Wi * CPSS_importance + We * CPSS_exec_time; //计算出任务的综合优先级
    if(ready_task->CPSS_policy == 0) //确定是否使用 CPSS 算法,当 CPSS_policy 为 0 时,则确定使用 CPSS 算法进行调度
    {
        while (ready_task->CPSS_compre_priority >= task->CPSS_compre_priority)
        {
            if ((task = task->rnext)->priority < 0) break;
        }
        task->rprev = (ready_task->rprev = task->rprev)->rnext = ready_task;
        ready_task->rnext = task;
    } // 比较任务的综合优先级,从链表中进行插入工作
    /* modify for CPSS algorithm end */
    ...
}
```

4.3 对定时器部分的改进

RTAI 中定时器有两种工作模式,分别为一次性模式(ONE_SHOT_MODE)和周期模式(PERIODIC_MODE).

由于此调度器采用的是 one_shot 模式的定时器中断机制,因此需要设置定时器的下一次中断时刻. 由于 RM 算法是经典的静态调度算法,因此需要调度在任务不会抢占当前正在调度的任务,而 CPSS 算法虽解决了 RM 算法经常错失周期长的任

务的缺点,但也是根据其综合优先级的大小排序来进行调度. 因此和 RM 算法一样的是,在当前运行任务的 possible_runtime 这个时间片中,是不允许另外任务抢占的. 因此定时器的中断时刻为 possible_runtime 时间片的末端时刻. 如图 1 所示.

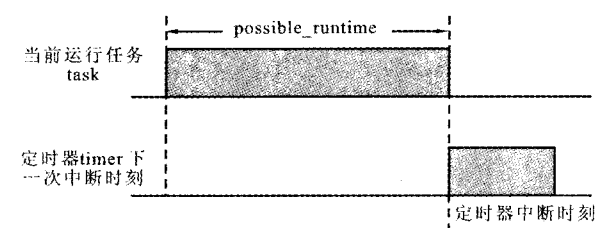


图 1 定时器下一次中断时刻图

5 RTAI 调度算法验证及其调度器的仿真

5.1 改进后的算法验证

5.1.1 算法对少数实时任务的验证

验证目标:测试 CPSS 算法是否针对 RM 算法更能使一些周期长但重要性高的任务尽快得到调度.

验证方法:通过给出任务的运行时间 Runtime 和重要程度 Importance 来计算任务的综合优先级 Compre_Priority. 对任务进行排序调度,与原先的 RM 算法的调度后的 DMR 进行比较. 由于 RM 算法在轻载情况下能达到较好的效果,则系统直接在重负载情况下进行测试.

验证指标 DMR(Deadline Missing Ratio):

截止期错失率(DMR)是衡量一个调度算法调度好坏性能的重要参数,其中错过其截止期的作业数为 n ,正常完成的作业数为 m ,则 $DMR = n / (n + m)^{[7]}$.

首先创建 4 个实时周期线程,具体线程参数如表 1 所示.

表 1 实时线程参数

任务	周期/ms	运行时间/ms	重要程度	综合优先级
P ₁	40	20	1	6.7
P ₂	40	15	3	6.6
P ₃	60	15	2	5.9
P ₄	60	20	4	8.8

任务的执行性能如表 2 所示.

验证结果:通过在重载情况下对 4 个实时线程的调度来进行对比,发现原本 P₃、P₄ 两个进程周期长,因此优先级较低,所以执行 RM 算法时执行性

能比较差. 而在 CPSS 算法中, 通过计算综合优先级, P_4 这个原本周期长的任务因为重要程度高而综合优先级高, 因此率先得到调度. 表 2 各项数据显示, CPSS 算法改进了 RM 算法对周期长的任务的调度差的缺点, 从而有效的改进了 RM 算法.

表 2 实时线程的平均执行性能

任务	RM 算法	CPSS 算法	DMR
P_1	100%	100%	RM 算法: 75.3%
P_2	100%	100%	
P_3	45%	83.3%	CPSS 算法: 34.2%
P_4	0	100%	

5.1.2 算法对大量实时任务的验证

因为 RM 算法和 EDF 算法在轻载时 DMR 都有较好的效果, 因此取系统负载 P 从 0.6 到 1.6. 比较新的算法在 RTAI 中响应 10000 个任务的截止期错失率, 具体仿真结果如图 2 所示.

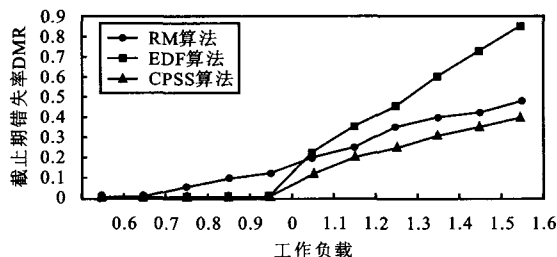


图 2 截止期错失率 DMR 比较图

仿真结果分析: 图中任务错失率方面, 静态调度算法 RM 算法在负载为 0.8 的时候第一个出现了错误, 而另外的两个算法在负载小于 1 的时候始终保持错失率为 0, 而当负载大于 1 之后, EDF 算法率先出现错失率陡升的情况, 而且 DMR 最高达到了 85%, 而 RM 算法虽然并未出现陡然上升的局面, 错失率却也不断上升, 而 CPSS 算法无论在负载小于 1 或者大于 1 的情况下错失率均在可承受的范围之内. 由此可见, CPSS 算法相较于 RM 算法和 EDF 算法是有改进的.

5.2 改进后的调度器仿真

实时系统的实时性能指标中最主要的指标是任务响应时间. 而测试调度的最主要指标是调度器时延. 所谓调度器时延是指求出的期望的调度点与实际调度点之间的时间差值^[6], 图 3 中显示了调度器时延的定义, delay 即为调度器时延.

测试结果:

测试时选取不同的 T 值, 分别测试了 $T = 100\mu s$, $T = 200\mu s$ 时的调度时延, 测试结果表明: 改

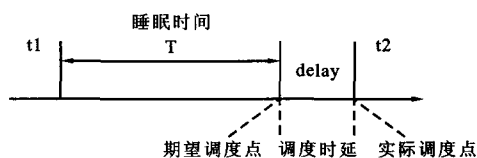


图 3 调度时延调度图

进后的 RTAI 调度器的调度时延在 $2\mu s \sim 7\mu s$ 之间, 大部分的调度时延集中在 $3\mu s$ 的时刻, 由此可以看出改进后的调度器调度时延相对于任务的周期值是非常微不足道的, 完全能够满足硬实时任务的要求.

综上所述, RTOS 本身的 RM 算法在任务错失率方面比动态算法 EDF 仍有较大差距, 在负载小于 1 时即出现错失, 而且容易错失任务周期长且重要度高的任务. 而 EDF 算法虽然是典型的动态调度算法, 但其缺点也明显, 即在强负载情况下错失率大大增加. 改进的 CPSS 静态调度算法较之 RM 算法增加了综合优先级的计算, 即使周期长且重要的任务更易得到调度. 且通过比较 DMR 和调度时延两个参数, 发现 CPSS 算法在调度算法和调度器方面均优于典型的 RM 算法.

6 结束语

文中分析了 Linux 操作系统的实施性改造方案—双内核 RTAI 的工作原理, 针对 RTAI 中的静态调度算法 RM 提出了一个改进算法 CPSS, 并重点针对算法设计了 CPSS 调度器及其利用程序将其实现. 通过仿真, 对 CPSS 调度器进行了仿真和调度时延的测试. 实验证明, CPSS 调度器可以在系统负载较重时仍然具有良好的实时调度性能, 适合硬实时调度的要求.

参考文献:

- [1] 张立辉, 赵云忠, 王建生. 基于嵌入式 Linux 的实时性分析[J]. 微电子学与计算机, 2007, 24(6): 100-103.
- [2] 汤子赢, 哲凤屏, 汤小丹. 计算机操作系统[M]. 西安: 西安电子科技大学出版社, 2001.
- [3] Raj Rajkumar, Kanaka Juvva, Anastasio Molano, et al. Resource kernel: A resource-centric approach to real-time and multimedia systems[C]// Proceedings of the SPIE/ACM Conference on Multimedia Computing and Networking. Pittsburgh, PA: ACM, 1998: 150-164.
- [4] Liu C L, Layland J W. Scheduling algorithms for multiprogramming in a hard real-time environment[J]. Journal of the ACM, 1973, 20(1): 46-61.

(下转第 155 页)

降低锌离子的活性,从而有效的抑制锌液残留,消除镍滋生,这两点有待于后继的实验分析来验证。

4 结束语

基于TSV技术的封装技术是目前MEMS产品、存储器、3D IC封装中的高端和热点技术。文中论述了基于TSV手机CIS芯片的晶圆级封装工艺流程及其先进性,研究了缩短该封装工艺流程和时间的一种方法,即将铝刻蚀、去胶提前到金属镀覆之前进行,这样光刻胶很容易去除,去胶时间短,不易残留,减小了去胶的难度,节约了时间,同时电镀工艺工序变少,使封装效率提高,节省了成本。

文中重点研究了铝刻蚀、去胶提前后镀镍时出现的镍滋生问题,对造成镍滋生的原因进行了分析,找出并验证了新工艺,实验获得了成效。该工艺稳定,被采纳列入正式项目,进入量产化的生产。

参考文献:

- [1] 郎鹏,高志方,牛艳红. 3D封装与硅通孔(TSV)工艺技术[J]. 电子工艺技术,2009,30(6).
- [2] 童志义. 3D IC集成与硅通孔(TSV)互连[J]. 电子工业专用设备,2009,(3):27-34.

- [3] 陈贵宝,阎山. 系统级封装技术现状与发展趋势[J]. 电子工艺技术,2007,28(5):273-276.
- [4] 杨光育,杨建宁,韩依楠. 电子产品3D—立体组装技术[J]. 电子工艺技术,2008,29(1):33-34.
- [5] Philip Garrou. 3D集成电路进入商业化领域[J]. 集成电路应用,2009(5):37-40.
- [6] 封国强,蔡坚,王水弟. 硅通孔互连技术的开发与应用[J]. 中国集成电路,2007,(3):54-57.
- [7] 練惠玉. 由TSV技术看CMOS影像感测器厂商布局[N]. 台湾区电机电子工业同业公会电子报,2008-10-15(41).
- [8] 刘圣迁,刘晓敏,张志谦,等. 微电子封装化学镀镍工艺研究及应用[J]. 电镀与涂饰,2005,24(1):40-43.
- [9] 李宁,袁国伟,等. 化学镀镍基合金理论与技术[M]. 哈尔滨:哈尔滨工业大学出版社,2000.
- [10] 曾华梁,吴钟达. 电镀工艺手册[M]. 北京:机械工业出版社,2000.
- [11] 刘巧明,夏传义. 用于微电子封装的化学镀纯镍研究[J]. 材料保护,1995,28(1):14-16.

作者简介:

董西英 女,(1961—),硕士,高级工程师. 研究方向为集成电路芯片工艺、封装测试。

(上接第150页)

- [5] 杨茂戴,紫彬. Linux2.6下内核调试技术的改进与研究[J]. 微电子学与计算机,2006,23(6):152-154.
- [6] 王创社,周树杰. RTAI实时调度器的优化与实现[J]. 北京石油化工学院学报,2007,15(3):51-55.
- [7] 王济勇,赵海,林涛,等. 定时器驱动的RM调度机制建模及其性能优化[J]. 计算机学报,2005,28(2):161-169.

作者简介:

李学桥 男,(1951—),教授,硕士生导师,CCF会员(E200008277S). 研究方向为嵌入式系统与智能控制、计算机网络。

梁爽 女,(1984—),硕士研究生. 研究方向为嵌入式与智能系统。

陈园 男,(1986—),硕士研究生. 研究方向为嵌入式与智能系统。