Linux操作系统实时性分析

余 兵,黎忠文
(厦门大学 信息科学与技术学院,福建 厦门 361005)

摘 要:随着Linux操作系统在嵌入式实时系统中的广泛应用,有效地提高Linux有限的实时性能是一个重要问题,而Linux内核可抢占调度是实时性能的改进的关键。对Linux内核调度器的工作原理进行了深入分析,并阐述了调度延迟是其实时性不强的原因,然后介绍通过可抢占机制对Linux内核进行改造,测试了改进后的内核的实时性。

关键词:Linux;实时性;可抢占内核;调度延迟
中图分类号:TP316.8 文献标识码:A 文章编号:1673-629X(2007)09-0041-04

Analysis of Linux Real-time Mechanism

YU Bing, LI Zhong-wen
(Computer and Information Engineering College, Xiamen University, Xiamen 361005, China)

Abstract: With the wide application of Linux operation system in embedded real-time system fields, the enhancement of Linux real-time performance becomes more and more important. Preemptive kernel is a decisive condition of a system's real-time performance. This paper analyzes of Linux scheduler and presents that the scheduler latency is the major cause of Linux OS' non-real-time. Then present a solution to enhance the real-time performance of Linux. In the end, the simulation results are represented.

Key words: Linux; real-time; preemptive kernel; scheduler-latency

0 引 言

Linux是一个性能卓越、技术上处于前沿的现代操作系统。但它是一个通用的分时操作系统,具有内核不可抢占、被动调度、优先级倒置、定时粒度粗糙等特点。

Linux内核的这些固有特点,使得Linux系统对实时任务的响应时间具有很大的不确定性和不可预测性。这样应用在实时性要求比较高的环境中,就有必要对Linux内核做进一步的改进。

Linux操作系统在实时系统领域面临这些挑战已经引起了业界的重视,目前已经有多种措施来提高Linux系统的实时性能。主要的策略有:

(1)增加实时子内核。如美国新墨西哥洲大学计算机系开发的RT-Linux[1],它是由两个子内核构成,一个用于Linux环境,一个用于实时环境。另外遵循GPL的RTAI[2](实时应用程序接口)也类似于这种方式。这两种方法可以有效改善系统中断延迟时间的问题。但这种策略设计了一个完全独立的实时核心而没有使用原有Linux内核,导致Linux系统的一些优势难以继承,尤其是与Linux内核相关的一些优势无法获得。比如Linux内核对大量硬件的广泛支持,Linux核心超群的可靠性、稳定性等。另外,由于这种方法并没有通过修改Linux内核代码来开发实时内核,而是在Linux系统之上重新设计了一个实时核心,这样的开发使得源代码不开放。

(2)为Linux打实时补丁[3]。这样可以借助Linux操作系统的源代码补丁来提高系统的实时性能。当前主要的实时补丁有低响应时间补丁、抢占任务补丁以及实时调度程序补丁等等。

综合考虑上面两种策略,通过修改Linux操作系统的源代码来提高系统的实时性更具有优势。

目前国内也已经有很多实时化改造的研究。但这些研究很多要么是只给出国外一些具有代表性的实时化改造方案[4],要么是从理论的角度笼统地介绍实时性的改造[5],都没有通过深入研究Linux源代码来分析Linux实时性的改进。对系统实时性的改造是一个很庞大的工程,面面俱到会导致分析不深入。因此在深入分析Linux有关进程调度和中断的源代码基础上,着重介绍通过抢占任务补丁来提高Linux的实时性。

收稿日期:2006-12-07
作者简介:余 兵(1982-),男,江西丰城人,硕士研究生,研究方向为实时操作系统;黎忠文,博士,副教授,研究方向为实时嵌入式系统的设计与开发。

1 Linux 操作系统实时性的瓶颈

1.1 实时操作系统的指标

首先来看实时操作系统对“实时”的衡量指标。一般来讲主要有两大指标,一是“中断处理延迟”,二是“任务调度延迟”。

(1)中断处理延迟。中断处理延迟是从中断产生到 CPU 开始响应该中断的时间段。最影响这个指标的是处理器关中断以及中断控制器屏蔽掉中断整个区域的代码量,无论是操作系统内核还是驱动程序还是应用程序(某些实时操作系统允许应用程序关中断),都会影响这个指标。另外影响这个指标的是硬件所产生的中断延迟,但这里所说的中断延迟只考虑软件产生的延迟,硬件产生的中断延迟一般只是通过提高硬件的处理速度来降低。

(2)任务调度延迟。任务调度延迟是指从系统需要进行进程调度到实际开始进行进程调度的时间段。一般来说,可以用当前运行低优先级任务的系统中出现了高优先级任务的时刻到高优先级任务被调度后开始运行的这段时间来衡量。这里主要关注任务调度延迟这个指标,因为 Linux 系统之所以“实时性”不好,很大程度上就是因为任务调度延迟太大。而影响这个指标最大因素是“内核抢占性”。对于 Linux 这样一个区分用户级和内核级的操作系统而言,如果内核不可抢占,那就意味着最大任务调度延迟指标会受到运行时间最长的那个系统调用时间的影响(原因在下文分析)。当然还有别的因素会影响任务调度延迟,如使用操作系统的虚存管理等。

总之,Linux 内核实时性不强,很大程度就是任务调度延迟过长,而导致调度延迟过长主要因素是内核不可抢占。

1.2 内核的不可抢占和关中断

Linux 2.4 内核的调度方式可以说是“有条件的可抢占”方式[6]。当进程在用户空间运行时,不管自愿不自愿,一旦有必要(例如,高优先级的进程已就绪),内核就可以暂时剥夺其运行而调度其他进程。可是一旦进程进入了内核空间,这时尽管内核知道应该要调度,但却不会发生,一直要回到用户空间的前夕才能剥夺其运行。下面结合 Linux 2.4 的原代码和特定情景来做进一步分析,此情景的执行过程如图 1 所示[7]。假设当前系统只有两个任务 A,B(理想化的假设),且任务 B 的优先级高于任务 A 的优先级。从 t0 时刻起,任务 A 在用户空间运行,任务 B 在睡眠,等待某事件中断唤醒。

t1 时刻,任务 A 要通过系统调用完成某个操作,这样,任务 A 就进入内核空间运行了。t2 时刻,某个事件发生引起外部中断发生(该中断处理程序会唤醒任务 B,这种情况在实时操作系统中很常见)。在内核中唤醒一个睡眠中的进程要调用函数 wake_up_process(),代码在 kernel/sched.c 中。下面是该函数中比较重要的语句:

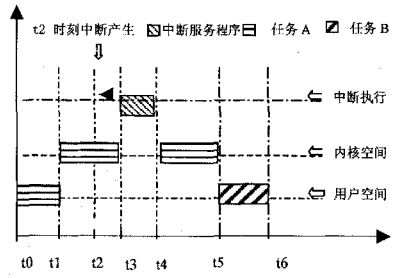

图 1 任务响应模型

```
p->state = TASK_RUNNING; //把进程的状态设置为 TASK_RUNNING
add_to_runqueue(p); //把该进程挂入到可执行队列中
reschedule_idle(p); //将所唤醒的进程 p 与当前进程进行比较,如果所唤醒的进程优先级更高,就将当前进程的进程描述符中的 need_resched 域置为 1
void reschedule_idle(struct task_struct * p)
{
    .....
    tsk = cpu_curr (this_cpu)
    //进程 tsk 是当前运行的进程
    if ( preemption_goodness ( tsk, p ,this_cpu) >1) //被唤醒的进程优先级高
    tsk -> need_resched = 1;
    //当前进程的 need_resched 置为 1
    .....
}
```

任务 B 的优先级高于任务 A,因此任务 A 的进程描述符的 neee_resched 被置为 1。

由于 t2 时刻系统在内核态下运行,CPU 可能关中断,所以中断不能马上响应。直到 t3 时刻,内核态代码使能中断响应(Linux 内核退出临界区),这个时候 CPU 开始响应在 t2 时刻产生的中断,并立刻开始执行中断服务程序;当中断服务程序执行完之后(在这个中断服务程序中唤醒了任务 B),中断返回。注意到此时返回的还是内核空间,此时不会发生任务调度,即使内核知道有更高优先级的任务需要调度。原因是 Linux2.4 中发生调度的必要条件是从系统空间返回到用户空间。因此,内核响应中断后,要到任务 A 的系统调用返回时才会发生调度,任务 B 才有可能被运行。

具体代码如下:

```
    ENTRY(ret_from_intr) # 中断返回
    ........
277 testl $(VM_MASK | 3), %eax
    #判断返回到用户空间还是内核空间
278 jne ret_with_reschedule
    #返回到内核空间不发生调度
279 jmp restore_all
    .......
217 ret_with_reschedule:
218 cmpl $0,need_resched(%ebx)
    #即使 need_resched 为 1 也不会发生调度
219 jne reschedule
```

在 t4 时刻,中断返回到内核空间继续执行任务 A,t5 时刻任务 A 系统调用完成以后返回,此时返回的用户空间,testl $(VM-MASK | 3), %eax 结果非 0,因此会跳转到 ret_with_reschedule 处引起任务调度。由于任务 B 的优先级高,在 t5 时刻发生调度后,任务 B 获得运行的资格。

在这个情景中,任务调度延迟时间是(t5 - t4),这段时间基本上等于中断返回后任务 A 系统调用的时间。所以最大任务调度延迟指标会受到运行时间最长的那个系统调用时间的影响。如果恰好任务 A 系统调用过长的话,那么调度就会过分地推迟,任务调度延迟时间会过大。而实时系统对响应时间有严格的规定,要求有很小的调度延迟。这样长的调度延迟对许多实时应用是无法满足的,这也是现实情况下优先级逆转的情况。

2　实时性的改进

Linux 内核不可抢占性既不满足实时调度算法的要求,也不能满足实时系统对外界事务响应延迟的要求,所以为了将 Linux 作为实时操作系统,提高它的实时性,必须让内核具有抢占性。

Linux 内核之所以被设计成不可抢占的,是为了保护内核的数据结构不被破坏,由此可以得出,如果内核可以抢占,必须提供一种保护机制用以保护内核的数据结构和临界资源。幸运的是为了支持对称并行处理器体系(SMP)设计的 spinlock 宏定义恰好符合这一要求。因而,内核可抢占方案的实现也就顺理成章了。

内核抢占的基本思想就是让调度程序获得更多的执行机会,从而减少了从一个事件发生到调度程序被执行的时间间隔,也就是降低任务响应延迟。在当前进程具有被"安全"地抢占的条件,并且有一个等待处理的重新调度请求,内核就会调用调度程序进行进程调度。那么什么情况下可以认为一个进程可以被"安全"地抢占?内核抢占要求内核中所有可能为一个以上的进程共享的变量和数据结构都要通过互斥机制加以保护,或者说都要放在临界区中。在抢占式内核中,认为如果内核不是在一个中断处理程序中,并且不在 spinlock 保护的代码中,就认为可以"安全"地进行进程抢占。为了让内核可抢占,需要对内核做如下三点修改[8,9]。

2.1　引入 preempt_count 计数器

为了支持内核抢占,为每个进程的 thread_info 引入了 preempt_count 记数器,该计数器初始值为 0。preempt_count 由宏 preempt_disable()、preempt_enable()以及 preemptenable_no_resched()所使用。preemptd_isable 对 preempt_count 计数进行递增,preempt_enable 对 preempt_count 进行递减。preempt_enable 宏查看当前进程的 preempt_count 和 need_resched 域的内容,如果 preempt_count 为 0 并且 need_resched 为 1,则调用 preempt_schedule()函数。该函数将给当前进程的 preempt_count 项增加一个很大的值(比如让 preempt_counter = preempt_counter + 0x4000000),然后调用进程调度函数 schedule(),在 schedule 函数返回后从该进程 preempt_count 中再减去该值。可抢占内核也修改了 schedule 函数,它检测进程的 preempt_counter 是否很大(这是为了屏蔽一些普通调度流程中对于抢占式调度来说是冗余的那些操作),然后执行抢占式调度。

2.2　更改 spinlock 宏定义

以前内核中的 spin_lock 自旋锁应用不同处理器对临界资源的竞争,而现在内核抢占要求应用于不同进程对临界资源的竞争。因而,要在 spinlock 宏定义中加入内核抢占互斥锁。在 spinlock 宏定义中加入了 preempt_disable()、preempt_enable()的调用使得内核抢占锁 preempt_count 和 spinlock 同步操作。

2.3　修改中断返回处理代码

它的修改主要在/arch/i386/kernel/entry.S 文件中。现在分析修改过的与中断返回有关的代码段。其中以 # 号开头的是对代码的注释。

```
00144:ret_from_intr: #从中断返回
00145:GET_THREAD_INFO(%ebp) #取得进程的 thread_info 信息
00146: movl EFLAGS(%esp), %eax # mix EFLAGS and CS
00147: movb CS(%esp), %al
00148: testl $(VM_MASK | 3), %eax #判断返回到用户空间还是内核空间
00149: jz resume_kernel # 返回内核空间
00150: ENTRY(resume_userspace) #返回用户空间
```

```
.....
00160: #ifdef CONFIG_PREEMPT
00161: ENTRY(resume_kernel) #返回内核空间的代码入口
00162: cli #关中断
00163: cmpl 0,TI_preempt_count(%ebp)# preempt_count是否为0
00164: jnz restore_all #不为0,返回,不抢占调度
00165: need_resched:
00166: movl TI_flags(%ebp), %ecx
00167: testb TIF_NEED_RESCHED, %cl #测试need_resched是否置位
00168: jz restore_all #need_resched没有置位,不抢占调度
00169: testl $IF_MASK,EFLAGS(%esp) # 如果关中断,则不允许抢占调度
00170: jz restore_all
00171: call preempt_schedule_irq #通知调度器目前这次调度是抢占调度
00172: jmp need_resched
00173: #endif
```

根据上面的三个修改可以看出,内核的抢占式调度发生在如下情况:在释放 spinlock 时,也即 preempt_count 为 0 时,当中断返回时,如果当前执行进程的 need_resched 被标记,则进行抢占式调度。内核可抢占后,启动调度器的条件放宽了很多,调度程序能尽可能多地运行。在前面举的例子中,从中断返回后,不要等到任务 A 的系统调用完成后才引发调度。而是在中断返回后就测试 preempt_count 是否为 0,如果为 0,则可抢占调度。而如果此时当前进程的 need_resched 被置位了,则会引发调度。即使中断返回后 preempt_count 不为 0,说明当前任务持有锁,不可抢占,但当当前进程(任务 A)持有的所有锁都被释放后,preempt_count 重新为 0。此时释放锁的代码会检查 need_resched 已被设置,引发调度,任务 B 获得运行资格。因此,任务调度延迟大大减少,系统的实时性得到增强。

3 实时性测试

对 Linux 作了实时化改造之后,往往需要对其作一定的测试来确定系统是否符合实际应用的需要。因此,如何衡量实时 Linux 的性能,也是实时化 Linux 中的一个重要问题。另外对 Linux 实时性能进行测试也可以发现 Linux 的性能特点,从而能进一步知道 Linux 实时化改造。这里最主要测试指标是任务响应时间,这是一个衡量 Linux 整体实时性的指标[10,11]。

3.1 测试原理

这里使用的测试工具是 realfeel。realfeel 是一个简单的用户程序,它的主要代码是 realfeel.c。它的工作原理是让测试程序对/dev/rtc 进行读操作,该读操作是一种阻塞读操作,它必须在一次实时时钟的中断后才能完成,因为这样才能得到最新时间。因而每次实时时钟产生中断,就可以结束一次对/dev/rtc 的读操作。程序设置实时时钟的中断次数为 2kHz,这样对/dev/rtc 的读操作也应有每秒 2048 次。因此每一次测量的理想值应为 1/2048 秒,把该理想值与实际测量值相比较就可得到系统的任务响应时间。

3.2 测试结果

测试系统的实时性能与硬件环境密不可分,此实验中的硬件环境是:2.26GHz CPU、256MB 内存。

首先在一终端运行程序./realfeel,并在另一终端运行命令 ping -l 1000000 -q -s 10 -f localhost 作为系统负载。在 kernel 2.6.15 的环境下得到的结果如表 1 所述。

表 1 任务响应时间(单位:ms)

响应时间	0.0	0.1	0.3	0.5	0.8	1.2	2.3	3.4	5.2
中断次数	286321	1095	210	82	16	8	3	1	1

很明显此时最大系统响应时间为 5.2ms。

在 kernel 2.6.15+可抢占补丁的环境下测试的结果如表 2 所述。

表 2 任务响应时间(单位:ms)

响应时间	0.0	0.1	0.3	0.5	0.7	0.8	0.9
中断次数	359884	1020	150	26	4	1	1

此时系统的最大响应时间仅为 300μs(0.3ms),可以看出,新构建的 Linux 内核的最大响应时间处于微秒级,比标准的 Linux 有了很大的改进。当然经过改造后的内核也只能满足软实时应用的需求,如果是要求严格的硬实时系统采用 RT-Linux 是较好的方案。

4 结束语

通过对 Linux 内核源代码以及内核实时性改进方案的了解和分析,更深入地阐述了标准 Linux 的实时处理能力以及有效提高 Linux 内核实时运算能力的途径和方案。最后对 Linux 实时性能测试工具与原理作了简单的介绍,并从测试结果可知,内核可抢占方案让 Linux 的实时性得到很明显的提高。

参考文献:

[1] Bbarabanov M. A Linux based Real Time Operating System [D]. New Mexico: New Mexico Institute of Technology, 1997.

[2] Moglen E. RTAI and the RT-Linux patent [EB/OL]. 2001-10. https://www.rtai.org/.

(下转第 47 页)

避障模块,也就是模糊控制器模块,该模块又包含了输入隶属度函数模块、输出隶属度模块和模糊控制规则模块。

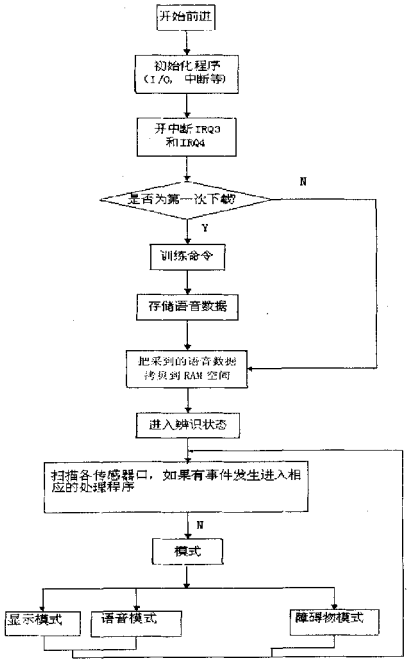

图 4　系统软件流程图

2.3 程　序

程序包括以下 3 个方面:

(1) 状态反馈。

(2) 前进和旋转运动控制。

(3) 多任务控制:用来设置各个行为的优先级,文中设置的优先级是这样规定的,避障行为优先级最高,显示模式显示小车的运动状态,音控优先级最低。

2.4 软件主要特色

软件在设计时,采用 C 语言和汇编语言的调用与嵌套。底层采用汇编语言编写,而在高层则采用 C 语言进行调用,这样不仅使程序更加简洁,同时也提高了程序的可靠性。使用模糊控制算法对人工智能进行了初步探索,可以预见,在软件上如果该算法和遗传算法、神经网络等理论有效地结合,在硬件上多种传感器有效地融合,小车的导航则可以进一步优化。

3 结　论

本智能小车功能要实施控制不是一个简单的电子控制问题,它涉及到光学、力学和机械学等领域,并与单片机相结合,从而使控制效果优化。文中创新点是:在检测路面障碍物时,硬件采用调制的脉冲驱动电路来驱动红外发射接收模块,红外接收电路采用 FPS6038,其内部已经集成了专用电路,具有调谐、放大、检测功能,增强了对抗外界光源干扰的能力;硬件中采用驱动单元、控制单元与检测单元相互隔离,信息通过光电藕合器来传递,提高了系统的稳定性;本系统硬件设计中在低功耗方面考虑周全,节省能源。软件中充分利用单片机的资源,加入特定人语音功能,使小车更加智能化;在避障方面采用模糊控制算法,使得小车的寻线性能和避障功能良好,控制灵活,基本符合要求。

参考文献:

[1] 罗亚非.凌阳十六位单片机应用基础[M].北京:北京航空航天大学出版社,2005.

[2] 方佩敏.新编传感器原理·应用·电路详解[M].北京:高等教育出版社,1994.

[3] 谢自美.电子线路设计·实验·测试[M].武汉:华中科技大学出版社,2001.

[4] 张懋兮.微型计算机(MCS－51 系列)原理、接口及其应用[M].南京:南京大学出版社,1999.

[5] 侯秀萍,袁秀丽,张　伟,等.模糊逻辑技术在医学诊断中的应用研究[J].微机发展,2005,15(5):94－96.

(上接第 44 页)

[3] Morgan K. A Response to Real Time and Linux, Part 3[M]. Santa Clara: Montavista Software Inc, 2002.

[4] 吴一民. Linux 实时化研究[J]. 计算机应用与软件,2003(1):9－10.

[5] 郭　强. Linux 实时性能增强技术的研究[J]. 微计算机应用,2005(4):481－484.

[6] 毛德操,胡希明. Linux 内核源代码情景分析[M]. 杭州:浙江大学出版社,2001.

[7] 杜　旭,胥海鹏. Linux 操作系统调度器实时性能的研究和改进[J]. 计算机工程,2005(5):100－102.

[8] 倪继利. Linux 内核分析及编程[M]. 北京:电子工业出版社,2005.

[9] Love R. Linux 内核设计与分析[M]. 北京:机械工业出版社,2004.

[10] 朱响斌,涂时亮. Linux 的实时性能测试[J]. 微电子学与计算机,2004(11):85－88.

[11] 周　云,徐佩霞. Linux 实时机制分析与改进[J]. 计算机应用,2003(5):81－82.