一种基于 Linux 操作系统下时钟系统的改进机制

唐红梅，郑 刚
（安徽工程科技学院 计算机中心，安徽 芜湖 241000）

摘 要：随着 Linux 操作系统的成功，改进 Linux 的设计和性能，使其应用于实时领域受到了越来越多人的关注。考虑到 Linux 目前的时钟粒度仍然粗糙，它将直接影响到整个内核的实时性能，文中提出了一种基于 Linux 操作系统下时钟系统的改进机制，找出修改时钟系统提高实时性的具体方法。实验结果显示，所作改进以不大的代价有效地提高了 Linux 的实时性能。

关键词：Linux；时钟系统；定时器队列；APIC
中图分类号：TP316 **文献标识码**：A **文章编号**：1673－629X(2007)02－0085－04

An Improvement Mechanism of Timer System Based on Linux Operation System

TANG Hong-mei, ZHENG Gang
(Computer Centre, Anhui University of Technology and Science, Wuhu 241000, China)

Abstract: Along with the success of Linux, how to improve Linux performance and to satisfy real－time requirement appeales more and more intension of developers. Because Linux timer granularity is granulated, it can affect real－time capability of entire kernel immediately. In this paper, put forward the improvement mechanism of timer system based on Linux operation system, find out the method of modifying timer system and enhancing real－time capability. Experiment result shows it is effective to improve Linux real－time capability by lower cost.

Key words: Linux; timer system; timer alignment; APIC

0 引 言

Linux 操作系统[1]中时钟系统的管理尤为关键，作为操作系统时间的唯一来源，时钟系统包括硬件和软件两个部分，直接影响整个内核的实时性能。一个良好的时钟系统除了具有准确性外还应该具有较高的精度，能够提供细粒度的时间刻度[2]。其次时钟系统本身运行所占的系统开销应该尽可能的少。在操作系统中，时钟精度不高的原因之一是因为周期性时钟中断的使用。操作系统不得不将大量的时间开销用于处理时钟中断，Linux 操作系统也是如此。

Linux 作为一种通用操作系统[3]，问题在于时钟粒度 1ms 对实时应用仍然粗糙，因此需要实现更高精度的时钟。目前常见的修改时钟系统达到实时化 Linux 方法都是从软件层面上着手，但是从时钟系统的硬件结构上分析并开展实时化工作也是一个值得注意的方向。文中，在分析了传统的 PC 时钟 8254 芯片实现细粒度时钟基础上，采用 APIC（Advanced Programmable Interrupt Controller）[4]先进的可编程中断控制器来实现定时功能。APIC 的时钟信号来自总线信号，采用 APIC 作为定时器，其时间精度可以达到纳秒级，从而减少改动获得较高的实时性[5]。

收稿日期：2006－04－13
基金项目：安徽省高等院校教学研究项目(2005262)
作者简介：唐红梅(1967－)，女，安徽芜湖人，实验师，研究方向为数据库与操作系统；郑 刚，副教授，硕士生导师，研究方向为操作系统与分布式计算。

1 定时器队列管理

定时器是 Linux 提供的一条双向循环定时器队列，所起的作用是在某个特定时刻唤醒某个进程完成工作，在系统中用一个全局变量 Jiffies 来表示。在应用中，定时器可分为三类：绝对定时器、相对定时器和循环定时器。定时器通过定时器控制块（TCB，Timer Controlling Block）来标识和管理，TCB 根据软件实现的具体需要可包括定时时长，指向同队列前、后一定时器的指针，申请定时器的进程标识等信息。

对定时器的组织和管理最简单的方式是采取先进先出（FIFO）方式的链表单队列，这种组织管理方式会

引起定时器队列太长、时间性开销很大等问题。文中针对这些问题给出了两种解决模型:定时器多队列分拆模型和循环多队列模型,以便能大大提高整个系统的性能。

1.1 定时器多队列分拆模型

多队列分拆模型主要解决定时器插入操作时间开销太大的问题。基本思想为:找到一种合理的分拆方式,把定时器按时长唯一对应到一个或多个按不同时长单位设置的子队列,该子队列按时长有序。每当时钟中断发生时,只对每一子队列的第一个 TCB 进行减法操作。要插入一个新定时器,先根据其时长分拆的结果,把该 TCB 首先挂到时长单位最短的子队列队尾,当该子队列对应的时长过去后,再取下重新挂到分拆的次短时长单位的子队列队尾,重复此操作直到定时器触发时间到。

算法描述如下:

(1)在 TCB 结构中增加对应的分拆子队列信息。

```
struct timerque {
struct TCB * heed, * tail ;// * 该子队列队头与队尾指针 *//
unsigned long eriterion ;// * 该子队列定时参照域,用于实现新定时器插入时相对时长的计算 *//
unsigned long timelong ;// * 该子队列对应的时长 *//
} TimerQ [MAX - 1] ;
```

系统初始化时,各子队列结构域中,队头与队尾指针皆为空,时长域值(timelong)按具体的拆分方式设定,定时参照域值(criterion)为该子队列的对应时长。

(2)时钟中断中定时器处理算法。

```
For(int i = 0 ;< MAX - 1 ;+ + )
  {while(TimeQ[i] 子队列不为空 and 队头结点的相对时长域值为 0 ;)
  取下 TimeQ[i] 队头结点,访问该结点对应的定时器控制块中的分拆子队列信息,判断是否还有时长更长的子队列,若有插入到下一个时长更长的子队列的队尾;否则触发该定时器。}
if(tq[i] 不为空)
  {tq[i] 队头结点相对时长域值减 1 ;
  tq[i] . criterion = (tq[i] . criterion + l) % tq[i] . timelong ; } }
```

1.2 循环多队列模型

基本思想为:分成 L 个定时器子队列,L 的选取原则应尽量使定时器子队列负荷均匀,最好取为 2 的指数,以便利用移位等操作计算倍数值和余数。基本数据类型为指向定时器控制块指针的一维数组 TimerQ 来集中这 L 个定时器子队列。TimerQ[i] 指向第 i 个定时器子队列的队头结点。如果所有定时器能均匀分布,则每个队列长度缩减为时长有序单队列长度的 $1/L$,从而大大节省了入队列时间。为了有效地减少时钟中断发生时的减法操作,使用一个循环计时指针 Current 指示当前队循环计时位置。算法描述如下:

(1)定时器插入算法描述:新的定时器应进入的子队列在 TimerQ 中索引 = (Current + 触发时间) MOD L,其中"MOD"表示取模运算。在该子队列中对新定时器按触发时间的先后进行排序插入操作,这时子队列中定时器的触发时间单位为 L 个时钟周期,即 L 的倍数时钟周期。假定 Current 当前值为 i,现有一触发时间长为 t 个时钟周期的进程需入队列,假定 k = (Current + t) MOD L,$d = t/L$,其中"/"为整数除法运算,插入后该定时器中的触发时间为 $d - 2 - 0 (d \geqslant 2)$。

(2)时钟中断对定时器队列的处理算法描述:每当时钟中断发生时,当前循环计时指针下移一位,即 Current = (Current + 1) MOD L 判断该 TimerQ[Current] 指向的子队列是否为空,如果为空,处理结束;如不为空则从该子队列头结点开始判断该结点所代表的定时器的触发时间是否为 0,如为 0 则表示触发时间到,触发该定时器,并从该子队列中删除;重复上一操作直至遇到第一个触发时间不为 0 的定时器或该子队列为空;若此时该子队列不为空,对队头结点触发时间值减 1,处理结束。

循环多队列模型与多队列分拆模型相比较,不需要对定时器的触发时间进行分拆操作,也不需要对超长定时器单独处理,从而能更好地提高系统性能。

由于在 Linux 定时器队列中都是按时钟周期的时钟粒度对各个进程的定时器进行处理的,无法判断更高精度的定时器,因此在内核中需要建立一个实时定时器队列,专门用于实时进程,即 APIC 实时定时器队列。当中断产生时,中断服务程序检查该队列中的所有定时器,同时从该队列中删除到期定时器。APIC 被重新编程,在下一个将要到期的时间触发 APIC 中断。为减少时间耗费,采用把 Linux 原定时器队列和 APIC 实时定时器队列结合起来处理的方法,提高效率。实现方法如下:把需要较长时间才到期的实时进程的定时器放入到原定时器队列中,借助该队列维护。当该进程的定时器快要到期时,8254 的周期性时钟中断处理程序会唤醒该定时器,此时将该定时器从原队列中移除,同时加入到 APIC 实时定时器队列中,避免了维护一个较大的定时器队列的开销,减少了定时器的迁移导致的排序查找等操作。

2 可编程时钟中断的实现

可编程中断定时器的主要作用就是从硬件上提供一个定时器,操作系统中的时钟中断一般就是由可编程中断定时器提供的。由于操作系统的运行离不开时

钟中断的驱动，因此可编程中断定时器是计算机不可或缺的一个部件。在 IBM 兼容 PC 机中，可编程中断定时器通常采用 8254 芯片，其端口号是 0x40－0x43。如果计算机只有一个 CPU，则可编程中断控制器采用 8259。如果采用多 CPU，则需要新的中断控制器，新的中断控制器根据 SMP 系统结构提供给每个 CPU 正确的中断信号。因此英特尔公司在现在 x86 计算机中采用了一个新的 I/O 可编程中断控制器芯片 APIC，用来取代 8259A 可编程中断控制器。许多 Intel CPU 都包含了一个当地 APIC（Local APIC），每个当地 APIC 包含了一个 32 位寄存器、一个内部时钟、一个当地定时设备以及两根中断线 LINT0 和 LINT1。所有的当地 APIC 都连接到一个 I/O APIC 上，这样就组成了一个多 APIC 系统。实现 APIC 可编程时钟中断步骤如下。

2.1　设置中断向量

Local APIC 集成在处理器内部，它为处理器实现了两个主要的功能：

1）它从处理器的中断引脚、内部中断源或外部 I/O PI C 接收中断，并且把它们发送到处理器核去处理。

2）在多处理器系统中，它发送并且接收来自其它处理器的处理器间中断，以便用来在系统中的处理器间发送中断或执行某些特定的指令。

由于 Local APIC 存在于处理器中，所以不能使用 request_irq()和 free_irq()函数来分配和释放中断。只能在 Linux 内核初始化的时候进行。内核进行中断机制的初始化主要调用 trap_init()和 init_IRQ()两个函数，其中 trap_init()函数主要是对一些系统保留的中断向量初始化，而 init_IRQ()函数主要用于外设的中断。Local APIC 中断向量的设置就是在 init_IRQ()函数实现的，这个函数在 arch/i386/kernel/apic.c 中。

条件编译选项为：

（1）CONFIG_X86_VISWS_APIC 是用于在多处理器情况下 APIC 中断机制初始化；

（2）CONFIG_SNP 用于设置跟多处理器有关的中断向量，如重新调度和关于 I/O APIC 的中断向量等；

（3）CONFIG_X86_LOCAL_APIC 用于设置 Local APIC；

（4）处理器内部中断 IPI（Inter－Processor Interrupt）用于设置 Local APIC 定时器中断、伪中断和错误中断；

（5）寄存器 APIC_ICR2 用来说明发送中断请求的目标，在单处理器中只能向自身 CPU 发送中断，在 18，19 两位设置为 01。

2.2　插入函数指针

（1）在 smp_apic_timer_interrupt()函数的前面加函数指针声明：

```
void (*apic_timer_handler)(void);//*设置 local APIC 定时器函数指针值*//
void set apic timer handler (void (*f)(void)) {//*给函数指针赋值*//}
```

（2）在 snip apic timer interrupt 函数中插入上述函数指针；

```
void smp_apic timer interrupt(struct pt_regs * regs)
  {int cpu=smp_processor_id();
   apic_timer_irqs[cpu]++;
   ack_APIC_irq();
   if (apic_timer_handler!=NULL) {
     apic_timer_handler}
```

（3）在 linux/include/asm－i386/apic.h 中的 Config_X86_Local_APIC 定义下加入

```
extern void set_apic_timer_handler (void (*f)(void));
```

（4）arch/i386/kernel/i386 ksyms.c 用于导出内核符号，在此文件头部加上 #include <asm/apic.h>，在 i386_ksyrns.c 中加入如下语句：export_symbol(set_apic_timer_handler)；

（5）使用 diff 命令生成 patch 文件，再用 patch 命令给内核打上补丁。

```
diff－Nur linux linux－6>linux－2.6.patch
patch－p()<linux－2.6.patch
```

2.3　设置 Local APIC 函数

设置 Local APIC 实时时钟中断函数包括：

（1）设置一次性定时模式：apic_setoneshot(APIC_LVTT，LOCAL_TNEIER_VECTOR)；

（2）设置周期性定时模式：apic_setperiodic(APIC_LVTT，Ox02e)。设置定时器中断时间需要向 Local APIC 初值计数寄存器中写入相应的数值。利用 apic_set(signed int clocks)函数设置 Local APIC 实时时钟粒度。

3　存在问题及解决方法

Intel 8254 PIT 芯片通过 IRQ0 产生周期性的时钟中断信号，作为操作系统的记时标准。在单 CPU 系统中，与时间有关的活动都是由 8254 时钟芯片来驱动的，8254 产生的是 0 号中断。HZ(include/asm－i386/param.h)参数代表了每秒中 8254 产生的中断数，即中断频率。该值在 Linux2.6 内核中设为 1000，内核一般通过 Jiffies 值来获取时间，因而默认的时钟粒度为 1ms，即时钟嘀哒（tick）。在 8254 时钟中断过程中，APIC 也可能触发中断，如果不响应 APIC 时钟中断，

则实时任务的请求不能得到响应,从而引起上下文切换等操作,增加系统开销。

解决方法:APIC 时钟中断响应期间关中断。由于 APIC 精度远高于 8254,因此当有实时进程的时候,可以在 APIC 时钟中断处理程序内部判断是否返回到 8254 时钟系统中,判断条件为下一个 APIC 实时任务的到期时间距现在的差值是否大于等于一个阈值,则返回到原时钟系统中,否则继续执行 8254 时钟。该阈值的确定与 CPU 速度有关,原则上等于执行一次 8254 时钟中断程序的时间加上下文切换时间,由于 8254 时钟中断返回时可能引起调度,因此还要考虑调度程序执行时间,把 T 统称为 8254 中断执行时间。

通过提高时钟频率可以改进时钟响应精度,但同时意味着原时钟中断又会被频繁地触发,系统开销加大,降低了整个系统的吞吐量,因此必须做出调节,防止这些情况的出现。

解决方法:设 $t1$ 为当前时间,$t2$ 为实时定时器队列中下一个任务的到期时间,last_ int_ time 为上一次 8254 时钟中断服务程序执行完的时间,T 为 8254 中断服务函数执行一次所需时间,max_ tick 为任意两次相邻 8254 中断相差最大时钟嘀哒数,伪代码如下:

```
void check_switch_time()
 {if((t2 - t1) ≥1ms + T)
   goto switch_to_8254;
   else if(((t1 - last_int_time) ≥1ms) &&((t2 - t1) ≥T));
   goto switch_to_8254;
if(((t2 - t1) < T) &&((t1 - last_int_time) ≥max_tick))
   Return;
switch_to_8254:
   local_irq_enable();//* 打开中断,切换到 8254 时钟系统中
*//
   timer_interrupt ();}
```

所有 x86 的微处理器芯片都包含了一个 CLK 的时钟输入引脚,用来接收外部晶振器的时钟信号。从奔腾芯片开始,许多近期的 x86 微处理器都包含有一个 64 位的时间戳计数器(TSC),该计数器的值能够被 rdtsc 汇编指令读出。TSC 对每个外部晶振信号进行加 1 计数,因此如果晶振频率是 1000MHz,则 TSC 每 1ns 进行加 1 计数。把 APIC 产生的时间计数值放入 TSC 中,但这种方法会增加从 CPU 周期到内存周期的 64 位时间值转换,需要对 APIC 进行编程。

Linux 中时间值用一个 32 位变量保存,即 Jiffies,作为整个时钟系统的标度,它存取效率较高。一个有效的方法是使用 32 位值保存 APIC 产生的时间,但是要考虑溢出问题。在 CPU 频率为 2G 的情况下,只要 1 秒钟 32 位计数值就会溢出。但由于采用了实时定时器队列的维护方式,仍可以采用 32 位数值来表示 APIC 时间,因为只需要对 1 个时钟嘀哒(1ms)内到期的实时任务进行处理,仅对当前时钟嘀哒开始的定时器进行高精度时间计数,该时钟嘀哒处理完以后,APIC 时钟计数值被清零,所以 32 位计数器足够处理。

采用 APIC 在内核中实现与原时钟系统并行的时钟系统具有良好的适应性。当无实时进程(即所有进程的响应时间≥1 个时钟嘀哒)时,APIC 时钟中断不会被触发,此时好像系统并未启用 APIC 时钟系统,与原来的时钟处理方式完全一样。当实时进程较多时,APIC 时钟中断被经常触发,8254 时钟中断被禁止,只有当 APIC 时钟中断执行完后,8254 中断才会被响应,即原有的周期性时钟中断方式变成了 one - shot 方式,8254 时钟中断只在 $t1$、$t2$ 和 $t3$ 三个时刻 APIC 时钟中断执行完成后才会被触发,在 10ms 内只触发了 3 次:分别是在第 2ms,5ms 和 9ms 的时候。

4 结束语

文中的设计目标是通过修改时钟系统提高 Linux2.6 内核的实时性,解决把具有分时操作系统特征的 Linux 改造成实时操作系统必须进行的时钟修改问题。为了实现这一目标,文中深入剖析了 Linux 时钟系统,找出修改时钟系统提高实时性的具体方法。试验环境采用 Intel Celeron 2G CPU,256DDR 内存,Linux Fedora 2.6.6 内核。结果 Linux 系统的中断延迟不超过 8μs,上下文切换时间不超过 19μs,任务响应时间在 30μs 以内。实验中的测试程序的主要工作只是记录运行时的状况,因此实验得出的数据比较客观地反映了实时性能指标。可以看出,改进后的 Linux 系统任务响应时间处于微秒级,与标准 Linux2.6 内核相比有了很大改进,在实时性能上已经达到了软实时操作系统的标准,可以支持大多数领域的实时应用,具有实际意义。

参考文献:

[1] 汤子瀛,哲风屏,汤小丹. 计算机操作系统[M]. 西安:西安电子科技大学出版社,2000.

[2] 陈莉君. Linux 操作系统内核分析[M]. 北京:人民邮电出版社,2003.

[3] 袁建红. 实用操作系统[M]. 北京:机械工业出版社,2002.

[4] 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.

[5] 李善平,刘文峰,王伟波,等. Linux 内核源代码分析大全[M]. 北京:机械工业出版社,2002.