第 27 卷　第 2 期　　计　算　机　学　报　　Vol. 27　No. 2
2004 年 2 月　　CHINESE JOURNAL OF COMPUTERS　　Feb. 2004

改善 Linux 核心可抢占性方法的研究与实现

赵慧斌　李小群　孙玉芳
（中国科学院软件研究所　北京 100080）

摘　要　随着开放源码的 Linux 应用逐渐普及，改进 Linux 的性能，使其适用于实时领域成为一个极具潜力的发展方向. 在参考了与此相关的研究的基础上，该文对改善 Linux 核心可抢占性的方法提出了 3 个改进措施：中断管理进程化、改进互斥锁的机制和增加互斥锁协议支持，并在 Linux 2.2 系列的核心上加以实现. 试验证明，这些改进达到了减少系统的抢占粒度，提高调度精度的目的.

关键词　Linux；可抢占性；实时；互斥锁；优先级反转；优先级继承协议
中图法分类号　TP316

Research and Implementation on Improving Kernel Premempablity of Linux

ZHAO Hui-Bin　LI Xiao-Qun　SUN Yu-Fang
（*Institute of Software*, *Chinese Academy of Sciences*, *Beijing* 100080）

Abstract　Using an opened source system, Linux, as the supporting OS is more and more popular in many fields, and it is regarded as a potential aspect by many users to improve Linux performance to satisfy the real-time requirements. Inspired mostly by some real time improvements of Linux, the article gives rise to three methods to improve the prempatibility of Linux kernel: tasked interrupt handling, modified mutex mechanism and mutex protocol support. The authors also describe the implementation under version of 2.2 series. The modification is a successful effort to decrease the grain of kernel trap and as a result, increase the precision of schedule.

Keywords　Linux; premeptibility; real time; mutex; priority inversion; priority inheritance protocal

1　引　言

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吸引了许多研究人员和开发人员的注意力，Linux 的实时化已有多种方案，如 RTAI[1]，RTLinux[2] 为代表的硬实时化，ART Linux① 等的软实时化方案. 我们在开发红旗实时操作系统 RFRTOS 的过程中，希望就核心的实时抢占性方面探讨一些改进措施，以增加 Linux 的调度粒度和准确性.

普通 Linux 核心是不可抢占的，这就是说，只有在核心作业完成后，才会发生调度. 不可抢占的核心设计具有一些优势，它使得互斥锁（mutex）[3] 的设计变得较为简单，核心数据的一致性容易维护，并且具有较好的稳定性；但作为实时系统的核心部分，可抢占性是一项较为重要的指标，它是决定系统调度精度的重要因素. 因而改进 Linux 的实时性，提高核心的可抢占性是充分条件之一.

ART Linux 在这方面提供了有益的尝试，它提

收稿日期：2002-03-19；修改稿收到日期：2003-04-08. **赵慧斌**，男，1974 年生，博士研究生，主要从事实时操作系统的研究以及基于 Linux 的嵌入式实时系统的开发工作. E-mail：hbzhao@sonata.iscas.ac.cn. **李小群**，女，1969 年生，博士，主要从事实时操作系统的研究以及基于 Linux 的嵌入式实时系统的开发工作. **孙玉芳**，男，1947 年生，研究员，博士生导师，主要研究方向为系统软件和中文信息处理.

①　ART Linux. http://www.etl.go.jp/etl/robotics/Projects/ART-Linux/

出了核心服务进程化的概念,改善了核心服务一体化造成的紧迫任务调度难于保证的问题,它实现了以下两种服务的进程化工作:

(1) 设备驱动进程化.

(2) 底半服务进程化.

ART Linux 提出了可抢占核心设计的思路,我们可以称之为核心服务细化的方案,但 Linux 在调度时机的选择上,粒度仍是核心服务级的,那么,采用 ART Linux 的方案仅是细化了抢占粒度,并未在根本上实现完全可抢占核心.我们认为改善核心的可抢占性,有两个主要的方向:中断管理的改进与互斥锁设计的改进.两方面彼此联系,相互影响.在参考了 ART Linux 的实时化方案的基础上,我们在开发 RFROS 中,提出了 3 个改进建议:

(1)中断管理进程化.

(2)强二元信号量的互斥锁机制.

(3)增加互斥锁协议支持,实现优先级继承协议[4].

本文详细阐述了这些方面设计的背景,给出了一种基于 Linux 2.2 核心的实现方法,并通过实验验证了这一改进的效果.

2　中断管理进程化

2.1　设计目标

中断管理的高效、可靠是系统设计的关键部分之一,设计不良的中断管理可能造成不必要的硬件等待,使得整个任务处理延滞,系统效率低下.Linux 是采用了宏内核的设计思路,这种设计理念下的操作系统易于实现,目前仍是一个重要的方向.但宏内核缺少分层的概念,层与层之间的服务或数据结构不能相互独立,而处于核心空间的中断服务又不可避免会直接操作硬件,于是用户的驱动程序的编程不容易约束,可能导致不合理的中断等待.

Linux 采用一种称作 bottom half[5] 的核心机制,可以部分改善其宏内核设计方面的一些缺陷,其主旨是中断处理的 top half 仅作一些必要的硬件操作,随后设置一些相关的标志位,在合适的时机,系统启动 bottom half,执行一些流程复杂的、耗时的但又不是十分紧迫的任务.一般而言,bottom half 的实质是唤醒与中断处理相关的进程.如果这种机制得到广泛应用的话,则中断处理将变得高效和易于维护.但遗憾的是底半的设计思想语义含糊,在系统中常常作为一种改进建议,而不是强制规范,所以用户完全可能绕过它,这使中断编程的随意性增大,效率难以保证,作为一种机制,它显得比较薄弱.

另一方面,Linux 中核心是不可抢占的.其原因是核心空间的作业被默认为具有最高的执行优先级,处于系统核心空间的中断或系统调用因而具有最高优先级.而这并不总是正确的,因为可能较低优先级进程通过系统调用进入核心空间,即使在此期间外部事件唤醒了某些较高优先级进程,它们也要等待低优先级进程的系统调用完成才可以被调度.

进程化的中断管理可以避免这些问题.进程化的中断管理使驱动程序由核心中一体化函数变为独立空间的进程,通过公开或屏蔽一些服务函数,中断服务进程使用的核心服务可以受到约束,不同的服务集合隔离了不同操作权限,实现了分层的设计理念.

另外,由于处理底层中断或其它核心服务的区域最小化,减少了核心空间的消耗,增加了抢占调度的精度.我们还设定每次中断返回都要检查是否需要重新调度,增强了进行任务抢占的时机.这样,改进的核心达到了以中断为粒度的可抢占精度,而不是 ART Linux 或 Linux 的核心服务的抢占精度.

中断处理进程化的另一个优势就是:中断处理优先级可以用来区分紧迫的中断处理和可以延后的中断处理,中断优先级的概念延伸到硬件,硬件就有了调度资源的一致描述.于是采用合理的中断调度将有助于区别高优先级的实时中断和低优先级的非实时中断,使得实时进程和非实时进程有了统一调度的可能.

2.2　实　现

Linux 提供了 kernel_thread 函数创建核心进程,核心进程具有 task_struct 描述和核心堆栈,共同参与进程调度;但它不同于普通进程,它运行于核心空间,没有用户空间,这意味着更高的执行效率,因为进程调度没有用户空间与核心空间的切换;另外,由于处于核心空间,可直接使用核心服务.

实现中断处理进程化需实现两方面的工作:中断注册和中断服务.中断注册 Linux 采用 request_irq 函数,在该函数中,主要是将用户指定的函数入口地址加入到全局中断描述数组 irq_desc 的服务函数描述成员中.RFRTOS 中,在此处做了部分修改,除完成前述工作,还要通过 kernel_thread 建立一个中断进程,该过程等同于如图 1 所示的功能代码.

```
for each hardware IRQ
do
  irq_desc[IRQ]action. handler := 
    IRQ_handler
  kernel_thread( IRQ_task, IRQ, 0 )
done
```

图 1 创建中断管理进程

中断管理的进程化使得中断服务在调度上"可见",这样,系统接受到中断后,并不是直接进入中断服务函数,而是通过设置调度标志告知系统即可,RFRTOS 在底层硬件响应中断的过程中,调用 wake_up 函数,唤醒相应的服务进程,如图 2(a)的功能代码所示,随后系统调度将在合适的时机选择执行服务进程.

由于中断服务贯穿整个系统生存周期,中断进程一般采用一个无限循环的形式,等待中断发生时调度执行.初始化时,中断服务进程通过一个 schedule 函数,放弃 CPU,系统选择其它进程.而当中断服务进程被硬件的中断响应唤醒后,则系统进入中断服务实体,这就是 request_irq 中注册的中断服务函数,一般完成一些如读写 I/O 端口等的实际服务任务.其功能代码如图 2(b)所示.

```
HardwareLayer:
do
    on each IRQ reached
    do
    wake_up(IRQ_task[IRQ])
    done
done
```

(a) 中断硬件层(top bottom)处理流程

```
IRQ_task (IRQ as Param):
do
    while TRUE
      schedule()
      irq_desc[IRQ]. action. handler( )
    end while
done
```

(b) 中断进程处理流程

图 2 处理流程

中断进程化的优点在于处于后台服务的中断管理调度上可见,系统的运作可控性较大,但也引起一定的调度开销,造成性能上的一些降低.我们为此在第 5 节的实验给出一个中断调度开销与原核心的比较,可以看到,这一时延仍在一定范围之内,且较为稳定,说明中断管理进程化的改进在效率上是可以接受的.

3 互斥锁

3.1 基于强二元信号量的互斥锁

Linux 中的互斥锁普遍采用了硬件锁的方式,互斥锁基于 spin_lock_irq /spin_unlock_irq 的原语引申,spin_lock_irq /spin_unlock_irq 在非 SMP 时,是如下所示宏定义:

```
_asm__volatile_("cli": : :"memory")
_asm__volatile_("sti": : :"memory")
```

其作用是关/开中断,这种互斥锁通过禁止中断,回避多线索引起的并发冲突,但这是代价非常高的一种互斥方案:任务只能交替执行.另外,该方法不能适用于多处理器的情况,当系统的处理器多于一个,就有可能有一个以上的进程同时执行,这种情况下,禁止中断依然不能解决问题.

在多处理器结构下,Linux 的 spin_lock 原语增加了 testset 指令集[6]的软件互斥实现,在一定程度上改善了互斥锁的效率,但其存在以下的明显缺陷:

使用忙等待:进程等待进入临界区,它处于忙等待,将消耗处理器时间.

可能出现饿死(starvasion):多个进程等待进入临界区,当临界区可用时,选择哪个等待进程是任意的,因此,某些进程可能无限等待.

死锁(dead lock)可能性较大.考虑这种情况:$P1$ 进入临界区 $S1$,中断发生更高优先级的进程 $P2$ 获得 CPU,$P2$ 希望获得 $S1$,互斥机制使其进入忙等待,但 $P1$ 比 $P2$ 的优先级低,不能获得 CPU 并放弃 $S1$,形成死锁.

基于以上的这些缺陷,我们在 RFRTOS 中采用了强二元信号量的互斥锁机制[7]的改进设计,实现了较为高效的互斥锁.同时,这种互斥设计易于实现复杂的互斥锁协议,如 Priority Inheritance Protocal(优先级继承协议),Priority Ceiling Protocal[4] 等,我们先给出以二元信号量实现互斥的伪码表示,如图 3 所示.

这种互斥锁机制属于"重量锁",它的实现是与进程调度相关的.算法采用了一个阻塞进程队列 queue,由于 wait 调用而被阻塞的进程将进入这一队列;当持有锁的进程退出关键数据区,发出 signal 调用,它在阻塞进程队列中选择一个被阻塞进程调度执行,在弱信号量的互斥锁实现中,进程的选择是没有策略依据的,因而可能出现某些进程饿死的现象,而强信号量的实现则依据一定互斥锁协议选择

进程,避免了前者的缺陷.

```
struct semaphore{
  enum state;
  queueType queue;
}
void wait(semaphore s){
  if(s.state==1)
  s.state=0;
  else{
    place this process in s.queue;
    block this process;
}
void signal(semaphore s){
  if(s.queue )
  s.state =1;
  else{
  remove a process P from s.queue;
  place P on ready list;
  }
}
```

图 3　强二元信号量的实现原语

3.2　实　现

我们在 RFROS 中的实现方法如下.

数据结构

基于信号量互斥锁结构 struct mutex_t:如图所示,state 含义同图 4(a)的 semaphore 的成员 state,它共有两种状态:LOCK_UNLOCKED,LOCK_BLOCKED;prochead 则同 task_struct 的成员构成了进程阻塞队列,prochead 顾名思义处于阻塞队列的队首,因而也是解除阻塞时应唤醒的进程.

struct mutex_t 中的 prochead 成员同 task_struct 结构中的 next_process_block,prev_process_block 成员把申请互斥锁而进入等待周期的进程组成队列,当互斥锁再次可用时,procehead 就会被选择执行,这就为强信号量的操作提供了支持.

```
struct mutex_t {
  int state;
  struct task_struct * prochead
  …
}
```

(a) mutex_t 的数据结构

```
struct task_struct{
  …
  struct task_struct * next_process_block;
  struct task_struct * prev_process_block;
  int lock_state;
  mutex_t * lock_blocked
  …
…
}
```

(b) 信号量相关的 task_struct 改进

图 4　数据结构及改进

```
lock ((mutex_t * lock){
  …
  <if (lock->state==LOCK_UNLOCKED) {
    lock->state=LOCK_BLOCKED;
    …
  }
  else {
    locking (lock);
  }>
}
locking (mutex_t * lock){
  …
  <addqueue(current, lock);
  current->lock_blocked=lock;
  current->lock_state=LOCK_BLOCKED ;>
  schedule()
}
```

(a) 互斥锁操作 lock

```
unlock(mutex_t * lock){
…
  <if (! lock->prochead) {
    lock->state=LOCK_UNLOCKED;
    …;
  }
else {
    unlocking (lock);
  }>
  if (current->need_resched) {
    current->flags |=PF_PREEMPTED;
    schedule();
  }
}
unlocking(mutex_t * lock){
    …
    adjust_runqueue (lock->prochead)
    …
}
```

(b) 互斥锁操作 unlock

图 5　操作算法

算法如图 5 所示.

实际的实现与伪代码相同,<>表示其中的操作为原子操作.

调度函数的改进

由于信号量互斥锁的引入,我们增加了一种进程状态:阻塞状态.在 task_struct 中的 state 和 lock_blocked 是描述与这一状态相关的成员.

schedule 函数中,与阻塞状态相关的部分我们加入了这样的处理(如图 6 所示):阻塞状态的进程将从运行队列中删除.与之相对的是放弃互斥锁的唤醒

操作，根据强信号量的定义，进程从阻塞队列中的移出是按一定顺序的. 因而，我们在随后的优先级继承协议实现中可以看到，必须按照协议唤醒相应进程.

```
if ( prev->lock_blocked …)
  del_from_queue(prev);
```

图 6 调度函数的调整

4 优先级继承协议

4.1 优先级继承协议描述

在实时操作系统中，常常面临的一个问题是优先级反转[4]. 优先级反转可以描述为：由于关键数据区的互斥锁保护，高优先级进程由于等待互斥锁阻塞而出现的低优先级进程处于运行态现象. 优先级反转使系统的行为难于预期，实时进程的时限不能得到保证. 而优先级继承协议可以解决优先级反转的问题，其算法的描述为：

(1)在任务 J 进入关键区间 CS，必须申请到互斥锁 S. 如果互斥锁已上锁，则 J 阻塞，否则 J 获得 S，进入 CS. J 退出 CS，必须释放 S，被 J 阻塞的最高优先级任务唤醒执行.

(2)任务 J 使用给定的优先级，直到在其互斥锁 S 保护的 CS 区间有更高优先级进程阻塞在 S 上. 在这种情况下，任务继承 PH，即被互斥锁 S 阻塞的进程中的最高优先级. J 退出 CS 后，恢复它进入 CS 时的优先级.

(3)Priority Inheritance 是有引申效果的；即如果 $J3$ 阻塞 $J2$，$J2$ 又阻塞 $J1$，则 $J3$ 通过 $J2$ 继承 $J1$ 的优先级.

优先级继承的互斥锁具有以下的特性：

(1)在优先级继承的条件下，任务 J 可能最多被阻塞min(n，m)个关键区间，在这里，n 是可能阻塞任务 J 且优先级低于任务 J 的任务个数，m 是可能阻塞任务 J 的互斥锁的数目.

(2)在优先级继承 Rate Monotonic 调度算法的条件下，当满足条件

$$\forall i, 1 \le i \le n, \sum_{k=1}^{i} \frac{C_k}{T_k} + \frac{B_i}{T_i} \le i(2^{1/i}+1),$$

我们称进程是可调度的. 其中 B_i 就是min(n，m)个关键区间的时间延迟.

优先级继承协议可以保证，在优先级继承的条件下，高优先级进程的延迟时间将在min(n，m)个关键区间之内，这使得实时进程运行的调度可以得到保证.

4.2 实 现

数据结构

为了实现优先级继承协议，task_struct 结构中加入了成员 rt_base_priority 和 locks_held，分别用于记录优先级继承前的优先级大小和记录进程持有的互斥锁，如图 7(a)所示. 改进 struct mutex_t 中，加入新的成员 owner，用于记录哪个进程占有当前的互斥锁，以便在优先级继承的实现中可以调整互斥锁拥有者的进程优先级. 如图 7(b)所示.

```
struct task_struct{
   …
   int rt_base_priority;
   …
mutex_t * locks_held;
   …
}
```

(a)PIP 相关的 task_struct 改进

```
struct mutex_t {
   …
   struct mutex_list_t{
   struct task_struct * proc;
   struct mutex_list_t * next, * prev;
   struct mutex_t * mutex;
   }mutex_list_t owners;
} mutex_t
```

(b)PIP 相关的 mutex_t 改进

图 7 PIP 相关的改进

主要函数

与优先级继承协议相关的函数是 inherit_prio 和 recalc_prio.

inhert_prio 函数的工作是在进程申请互斥锁被阻塞时，根据优先级继承协议更改持有锁进程的优先级，为了代码清晰，我们采用了递归的描述方式，如图 8(a)所示.

而 recalc_prio 函数则正相反，是在进程放弃互斥锁时，恢复初始的优先级，同样采用递归描述，如果进程优先级曾调整过，则需要恢复原优先级，包括进一步调整可运行队列，如图 8 (b)所示.

```
inherit_prio(task_struct * proc){
  mtx=proc->lock_blocked;
  priority=proc->rt_priority;
  if(mtx->state==LOCK_BLOCKED){
    p=mtx->owners->proc;
    if(p≠∅ && p->rt_priority < priority){
      p->rt_priority=priority;
      if(p->lock_blocked≠∅){
        ajust lock queue of p in a desendent
          order of process priority
        inherit_prio(p);
      }
      else {
        …
        wake_up_process (p);
      }
    }
  }
}
```

(a)优先级继承函数

```
recalc_prio(task_struct * proc){
  …
if(proc->rt_priority==proc->rt_base_priority)
    return;
  …
  priority=proc->rt_base_priority;
  …
  lowered=proc->rt_priority - priority;
  proc->rt_priority=priority;
  if((mtx=proc->lock_blocked) ==∅){
    adjust_runqueue(proc);
    return;
  }
  ajust lock queue of proc in a desendent order
    of process priority
  p=mtx->owners->proc;
  if(p≠∅) recalc_prio(p);
}
```

(b)优先级恢复函数

图 8　优先级的继承与恢复

5　实　验

5.1　中断进程化的调度开销

RFROTS 的中断管理是进程化的,这样,中断处理除受到中断禁止的影响,还可能要经过调度选择和进程文境切换的过程,这些开销将会影响中断响应的效率,为了测试中断线程化改进对系统效率的影响,我们在系统中增加监测点,并利用注册 proc 文件系统的方式回显测试结果,在每个监测点上,我们都记录硬件中断触发时 TSC 时标和记录进入相应中断线程的 TSC 时标,利用二者之差,我们可以得到中断延迟的时间.

比较原核心和实时核心的中断处理时延,得到如图 9 的统计结果.

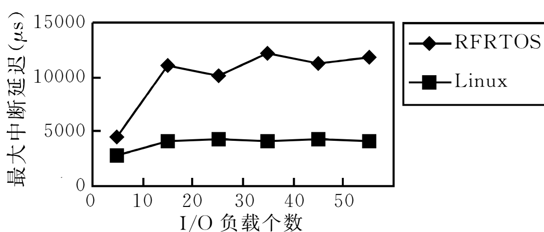


图 9　中断管理改进开销

在未经改动的 Linux 核心中,中断延迟主要是由于中断禁止所引起的,因而中断响应会受到负载、尤其是 I/O 负载的一些影响,但随着负载的增加,响应延迟趋于平缓,最大的延迟稳定在 4μs 左右,这体现 Linux 中断处理设计效率方面的合理性;而在 RFRTOS 中,我们看到,统计数据中,RFRTOS 最大的中断延迟为 12μs,对效率的影响仍在可以接受的范围之内;另外,由于影响较为显著的是文境切换和调度效率,负载的增加并未给中断延迟带来直接的影响,因而延迟的稳定度也较高.

5.2　优先级继承

在我们的试验中,建立了 3 个进程,以一个数组 J 表示,数组下标对应进程的优先级,我们这样定义一个操作:在时刻 t,任务 J 对锁 CS 执行了一个动作 Op,可描述为

$$\{t:J.Op(CS)\},$$

其中 Op 是 $Lock$ 或 $Unlock$ 之中的一个操作,分别表示互斥锁上锁和解锁.

试验中,我们进行了这样的操作序列:

$$\{t_0:J_1.Lock(CS1)\}\rightarrow\{t_1:J_2.Lock(CS2)\}\rightarrow\{t_2:J_2.Lock(CS1)\}\rightarrow\{t_3:J_3.Lock(CS2)\}\rightarrow\{t_4:J_1.Unlock(CS1)\}$$

$J1$ 的代码中,包含一段监测和记录其优先级变化的流程,试验得到如图 10 所示的结果.

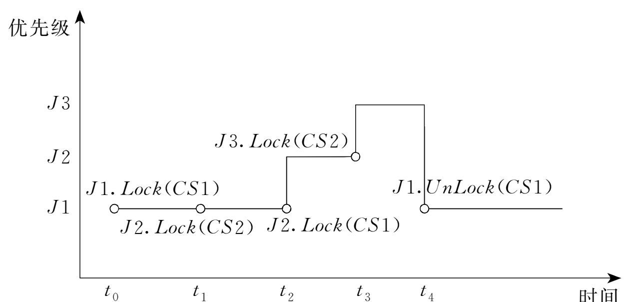


图 10　优先级继承下的进程优先级变化

与优先级继承协议相印证,不难看到 J_1 在 t_2 时刻继承了 J_2 的优先级,随后在 t_3 时刻,又通过 J_2,继

承了 J_3 的优先级.该试验证实了优先级继承协议在 RFRTOS 的实现的正确性.

5.3 可抢占核心

为测试核心的可抢占性,考虑一个系统调用模型:busy_sleep,该系统调用忙等待一段时间(实验中设为 4s),用于模拟一个较为耗时的系统调用.功能代码描述为:

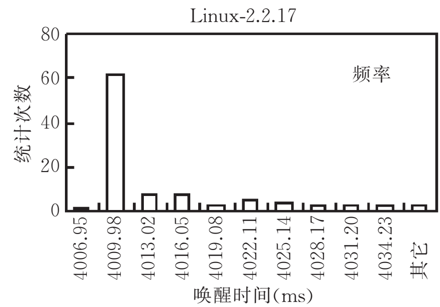
```
asmlinkage void sys_busy_sleep()
{
  busy sleep for 4 second
}
```

另外,以下的用户测试程序将表现出该系统调用对任务抢占所造成的影响:

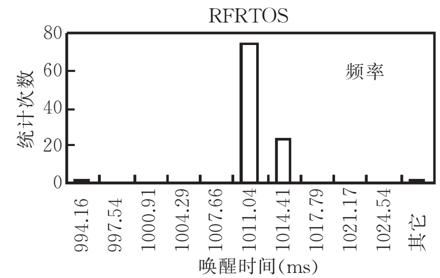
```
Task1:
do
  repeat 500 times
  do
    sleep for 1 s
    record current time
  done
done
Task2:
do
  repeat 100 times
  do
    call system call busy_sleep
  done
done
init:
do
  threads is type of array of thread
  create_thread (threads[0],Task1)
  create_thread (threads[1], Task2)
  join_thread(threads[0])
  join_thread(threads[1])
done
```

Task1 调用 sleep 睡眠 1s,Task2 则调用系统调用 busy_sleep,通过监测 Task1 的唤醒周期,可以比较核心调度可抢占性方面差异.我们分别在未改动的核心 2.2.17 和 RFRTOS 上做了测试,结果如图 11(a)和图 11(b)所示.

由图 11(a)可以看到,在未改进的核心上,虽然我们设定 Task1 每隔 1s 被唤醒一次,但 Task2 由于调用了 busy_sleep,阻塞了 Task1 的执行,实际的情况是 Task1 在 4s 后才能被调度执行,这也正是 busy_sleep 的延迟时间,也就是说,一旦某个进程陷入核心态,其他进程不能被调度执行.



(a) Linux-2.2.19 试验统计



(b) RFRTOS 试验统计

图 11

而由图 11(b)看到,在 RFRTOS 中,Task1 的唤醒执行时间则较为准确地集中于 1s,证明核心服务 busy_sleep 并不能阻塞其他进程的调度需求,这体现了核心可抢占性方面有了较为明显的提高,进一步说明了改进的合理性.

6 总 结

本文给出了改进 Linux 可抢占内核设计的基本思想,并在此基础上实现了一个可用的改进核心——RFRTOS,提升了核心可抢占性,提高了调度精度,它作为红旗 Linux 公司的产品,正处在市场推广阶段.目前,该系统只实现了优先级继承协议,以后的工作将包括实现 Priority Ceiling Protocol,并在加强系统的稳定性方面进行一些探索.

参 考 文 献

1 Cloutier P., Montegazza P., Papacharalambous S., Soanes I., Hughes S., Yaghmour K.. DIAPM-RTAI position paper. In: Proceedings of the 2nd Real Time Linux Workshop, Orlando, FL, 2000,122～131

2 Barabanov M.. A Linux-based real-time operating system. New Mexico Institute of Mining and Technology[M.S. dissertation]. Socorro, New Mexico, 1997

3 Tanenbaum A.S., Woodhull A.S.. Operation Systems Design and Implementation. Second Edition. New Jersey: Prentice-

Hall, 1997

4 Sha L., Rajkumar R., Lehoczky J.. Priority inheritance protocols: An approach to real-time synchronization. IEEE Transactions on Computers, 1990, 39(9):1175～1185

5 Mao De-Cao, Hu Si-Ming. Linux Kernel Source Analysis in Scenario. First Volume. Hongzhou: Publishing House of University of Zhejiang, 2001(in Chinese)
(毛德操,胡希明. Linux 内核源代码情景分析. 上册. 杭州:浙江大学出版社,2001)

6 Lamport L.. The mutual exclusion problem: Part I and II. Journal of the ACM, 1986, 33(2):313～348

7 Stalling W.. Operating System. 4th Edition. Beijing: Publishing House of Electronic Industry, 2001(in Chinese)
(Stalling W.. 操作系统——内核与设计原理. 第4版. 北京:电子工业出版社, 2001)

ZHAO Hui-Bin, born in 1974, Ph. D. candidate. His primary research interests include research of real time operating system and development of real time OS based on Linux.

LI Xiao-Qun, born in 1969, Ph. D.. Her research interests focus on the real time operating system.

SUN Yu-Fang, born in 1947, professor and Ph. D. supervisor. His research interests focus on the processing of the system software and Chinese information.

Background

The paper is mainly focused on the topic of real-time enhancement of Linux, an open sourced operating system, which is a major research direction for many developers as long as Linux becomes a more and more popular supporting OS for many architectures and applications. But the design philosophy of Linux takes little care of the requirement for applications with real time constraints and as a result, such concerns as kernel premempablity and resolution of system services must be considered to improve the real time property of Linux.

The research has some similarities to KURT and ART-Linux, which are soft real-time enhanced Linux projects and have advantages over the other well-known hard real-time Linux projects, RTLinux and RTAI, in such aspects as fully available system services, completely functional driver support and flexibly alternative schedule mechanisms. KURT tries to improve Linux real-time property by enhancing the timer resolution but has little concerns on preemptive scheduling, thus leading to a coarse system responsive granularity. On the other side, ART-Linux takes little care about real time scheduling mechanism though it makes effort to build a preemptive kernel and a rather rough priority inheritance protocol.

This paper gives rise to a mechanism, tasked interrupt services, through which preempting at hardware interrupt level is achieved. Implementations of semaphore are described, too, to improve the mutex design of Linux and based on that, an effective PIP is brought forward. The modifications prove to be a successful effort to decrease the granularity of kernel trap and as a result, increase the precision of schedule for real time applications.