# Linux 互斥锁机制的研究及改进

赵慧斌　李小群　孙玉芳　叶以民
（中国科学院软件所一部　北京100080）

## Reasearch and Improvement on Mutex of Linux

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

**Abstract** Applications with real-time constraints are not only growing in the field of embedded system, but gaining popularity in the desktop environment as well. At the same time, using an opened source system, Linux, as the supported OS is more and more appealing to many developers. So it is regarded as a potential aspect by many users to improve Linux performance to satisfy the real-time requirements. The article discusses the mutex implementation of Linux in depth and on the basis of that, gives rise to methods to improve the deficiency. An implementation under version of 2.2 series is brought forward.

**Keywords** Realtime, Mutex, Priority inheritance protocol, Priority inheritance, Linux, SMP

## 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，KURT[3]、ART Linux[4]等的软实时化方案。

硬实时化方案主要在中断处理、调度和时钟管理方面进行了改进工作，中断处理方面，RTAI 提出的中断虚拟层[1]减少了由于中断屏蔽而引起的调度延迟，硬实时化方案同时提供了一个具有进程管理、中断管理等基本核心服务的实时核心为实时任务服务，而原核心对于该实时核心来说是一个 idle 进程。另外，硬实时化方案采用了动态精度的时钟管理策略，既保证了实时任务对时钟精度的要求，又减少了由于时钟中断过于频繁而引起的不必要的系统负载。硬实时化方案可以达到微妙级的中断和调度响应，但由于采用了小实时核心提供实时服务，其服务范围受到了较大限制。

软实时化方案主要以 KURT 和 ART-Linux 为主，KURT 的目的在于满足在 ATM 及多媒体方面的研究工作需求。这些研究工作对操作系统提出了特殊要求，其特殊性在于：既要求有很高的实时性，如与时间相关的 QoS 保证，又要求全面的操作系统服务。分时系统无疑可以满足后者，但无法满足前者，即便实现了 POSIX1003.1b，其实时性依然显得太弱；而专用实时系统可以满足前者，对于后者却又无能为力。

KURT 对 Linux 核心做了如下两点改动：

1)修改了时钟中断机制，采用变长时钟精度的方法。

2)增加了新的实时调度模块。

KURT 并没有对核心做更多的改动，因而在核心可抢占方面并无本质提高，这限制了其应用的广泛性，KURT 实质上是一个弱实时系统。

ART-Linux 则注重在改进核心可抢占性方面的提高，它在改进核心互斥机制和 Linux 核心服务细化方面做了较大改进，这与我们的改进工作较为接近。但 ART-Linux 在核心互斥机制方面的改进较为简单，不支持读锁。

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。在不损失系统服务的情况下，核心的可抢占性成为决定实时性能较为重要的一个条件，而 Linux 设计欠缺这方面的考虑，体现在互斥锁[5,6]设计方面有以下两方面的缺陷：

互斥锁设计较为简单，难于保证互斥效率，不能解决优先级反转问题。

本文详细讨论了 Linux 互斥锁的设计，并在此基础上提出了一种改进 Linux 的互斥锁设计的办法——基于强二元信号量[7]的互斥锁。基于强二元信号量的互斥锁不仅在互斥质量上优于其他类型的互斥锁，更为重要的是，基于改机制，可以实现优先级继承协议（Priority Inheritance Protocol，PIP）[8]和优先级冲顶协议(Priority Ceiling Protocol，PCP)[8]，增加系统的可调度性。

## 2 Linux 的互斥锁

Linux 的支持多种互斥锁，总的来说，包括基于中断禁止/启动硬件锁，基于 test and set 指令集的互斥锁和基于信号量的互斥锁。这三种互斥锁有各自的适用范围，而基于信号量的互斥锁在 Linux 的设计较为简单，应用范围相对狭窄，我们这里简要介绍前两种互斥锁的实现。

### 2.1 硬件互斥锁

单处理器下，Linux 可以通过简单的开关中断，完成互斥保护。这种互斥锁较为简单，通过禁止中断，回避多线索引起的并发冲突。这种互斥锁的设计具有一些优势：

设计简单，不依赖进程环境。

由于从硬件上限制了多线索，回避了操作系统层面的死锁问题。

但同时，硬件锁的代价非常高：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。

### 2.2 基于 test and set[7]指令集的互斥锁

Linux 针对 SMP 的设计中，使用 test and set 指令；单处理器的情况下，简单地用 cli 或 sti 语句可以获得互斥效果，但对多处理器是无效的，因为即使中断禁止，多个任务同时在多个处理器运行，依然不能保持互斥条件；而且，在 X86结构下

的 SMP(Symmetric Multiple Processor)的 APIC(Advanced Programmable Interrupt Controller)[9]的机制下,cli 或 sti 语句成为一种本地化的操作,并不能保证改变 SMP 的多线索这一事实,因而也不能确保并发冲突。

X86的架构设计中,原子操作在操作系统内存的单指令过程中是默认的,因为这种指令要获得内存总线锁后才可以执行;但有些数据被 cache 处理后,这种默认并不能保证数据的原子操作,因为某个处理器进行一个指令可能是在它本地的 cache 中进行的,所以协调 SMP 的原子操作,还要另外的机制;Intel P6系列是通过在汇编语句中加 lock 前缀获得的,lock 前缀可以保证内存或 cache 的操作的数据一致性。test and set 指令集采用这种原子操作,保证进入关键数据区的操作的唯一性。

基于 test and set 指令集的互斥锁在 Linux 中,采用了 spin_lock 的原语描述,可以表示为如下所示的伪码,伪码表示中,〈〈〉〉所包围的代码是受 lock 保护的;另外,文中其他部分,如果某处代码在 SMP 下是原子操作,统一用〈〉标示:

```
spin_lock
do
1:
    if 〈〈testandset(flag,0)〉〉then
        critical section……
    fi
    repeat while flag
    do
    done
    goto 1
done
```

图1 SMP 下的 spin_lock 原语

Linux 为了保证 SMP 的效率,还提供了 read_lock 和 write_lock 两种锁机制,它用了一个32位的整形数 rwcount 作为标志,最高位是读锁标志,一个读锁可以同时容纳$2^{31}-1$个不同线索进入,我们给出其功能代码,如图2所示。

```
readlock
do
    1:
        〈〈rwcount:=rwcount+1〉〉
        if rwcount>2^31-1 then
            goto 2
        else
            critical section……
        fi
      2:
        〈〈rwcount:=rwcount-1〉〉
        repeat
        while rwcount>2^31-1
        do
        done
        goto 1:
done
```

图2(a) SMP 下的读锁

```
write_lock:
do
1:
    if!〈〈testandset(rwcount,31)〉〉then
        goto 4
    fi
2:
    if 〈〈rwcount>0 AND rwcount<2^31〉〉 then
        goto 3
    critical session…
3:
    if 〈〈testandset(rwcount,31)〉〉 then
        goto 1
    fi
4:
    repeat while wcount
    do
    done.
    goto 1
done
```

图2(b) SMP 下的写锁

test and set 指令集适用性广泛,实现简单,可以在无上下文文境的环境下应用,和以下提到的基于信号量的"重量级"互斥锁相对应,被称为"轻量级"互斥锁,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(starvation):多个进程等待进入临界区时,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获得处理器,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,不能获得处理器并放弃 S1,形成死锁。

## 3 基于强二元信号量的互斥锁改进

Linux 提供了基于信号量的互斥锁的支持,但是,它的实现过于简单,存在一些问题:

1)没有提供优先级反转的解决办法,这一点需要增加一些互斥锁协议,本文将详细讨论。

2)采用了弱信号量的实现方式,某些进程可能会一直等待,出现饿死的情况。我们实现的支持 PIP 的互斥锁,其本质就是一种强二元信号量的互斥锁,因而也就解决了这一问题。

3)使用范围较为狭窄。基于信号量的互斥锁只能在进程文境下使用,而 Linux 属于宏内核设计,没有进程文境概念的系统空间范围较大,所以不能普遍使用这种锁机制,仅在内存管理和文件系统等有一些较为局限的应用。

出于设计方向的原因,Linux 互斥锁的实现较为简单,虽然可以提供互斥支持,但效率方面并不理想,在桌面或服务器应用中,这方面的问题并不突出,但对实时应用,该设计将显著影响系统的性能。

为增加系统的实时性能,互斥锁的改进思路不外以下几点:

1)以软件锁代替硬件锁,软锁不但可以减少因中断屏蔽引起的系统延迟,还可以适用于 SMP 的情况。

2)防止优先级反转,支持多种互斥锁协议。

从互斥锁实现和协议支持能力,显然基于信号量互斥锁可以较好地解决,我们的改进思路正是这样的。改进中,我们面临以下问题:基于信号量的互斥锁需要进程文境,而 Linux 是宏内核的设计,中断空间、底半处理和信号处理都没有自己的进程文境,难于实现信号量互斥。另外,Linux 本身的信号量互斥锁实现较为简单,并不能解决优先级反转的问题。

关于宏内核的设计问题,我们采用了 ART-Linux 所提供的核心服务进程化思路,将中断服务、底半处理和信号处理改为核心进程,使其具有独立的进程文境。本文集中于互斥锁的支持,核心服务进程化在此不再赘述。

在核心服务进程化的改进之下,我们扩展了信号量互斥锁的使用范围,取代以前的基于 test and set 指令集的互斥锁在改进的互斥锁基础之上,可以实现针对实时任务的改善优先级反转的问题,为此我们增加了互斥锁结构,并在进程描述结构增加了与此相关的新成员。

如图3(a)所示,state 表示是互斥锁的状态,共有三种状态:LOCK_UNLOCKED, LOCK_WRITE, LOCK_READ;num_readers 是持有该锁的读进程数目。prochead 则构成了进程阻塞队列;互斥锁协议的选择由 proto 决定。owners 保存

拥有锁的进程信息，一个锁可同时被多个进程获取，拥有锁的进程就是 owners。可以看到，其成员包含了一个持有该锁的进程 proc，和记录持有锁的所有进程的链表成员 next 和 prev。

```
struct mutex_t{
int state;
    int num_readers;
    struct task_stuct * prochead,…
    int proto;
    mutex_list_t
        owners [CONFIG_MAX_READERS];
}
struct mutex_list_t{
    struct task_stuct * proc;
    struct mutex_list_t * next, * prev;
……
}
```

图3(a)　mutex_t 的数据结构

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

图3(b)　信号量相关的 task_struct 改进

进程的 task_struct 也做了必要修改，如图3(b)所示，增加 rt_base_priority 成员是进程未发生 PIP 的优先级，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联成一个队列。locks_held 是该进程持有的锁，相对应的 lock_blocked 是使该进程阻塞的锁。

在互斥锁的实现中，采用了读锁和写锁两种操作，写锁是独占锁，同一时间，只有持有锁的进程可以访问关键区间，而读锁则允许多个线索同时进入关键区间，在 RFRTOS 设定可以同时进入关键区间的读进程数目为16个。相关的互斥锁操作主要有 write_lock 函数和 read_lock 函数，我们一一介绍。

write_lock 算法

```
参数:lock is type of struct mutex_t
do
    current 为当前运行的进程
    (if lock. state=LOCK_UNLOCKED then lock. state:=LOCK_
        WRITE
        将 lock 加入到 current. locks_held 列表中
    else
        do_locking(lock,LOCK_WRITE)
    fi)
done
```

read_lock 算法

```
参数:lock is type of mutex_t
do
    current 为当前运行的进程
    (if lock. state≠LOCK_WRITE
            OR
        lock. num_readers≠最大读进程数)
        …
    then
        获取一个 lock. owners 的节点
        将该节点加入到 current. locks_held 列表
        lock. num_readers++
        lock. state:=LOCK_READ
    else
        do_locking(lock,LOCK_READ)
    fi)
done
```

与申请锁相对应的是解锁操作，分别由 write_unlock 和 read_unlock 完成，由于逻辑上的对应关系，我们这里无需赘述。

read_lock 和 write_lock 最终通过 do_locking 完成整个的阻塞流程，在设定进程相应的状态后，调用 schedule 函数进行必要的进程切换。

do_locking 算法

```
参数:lock is type of struct mutex_t,lock_type of int
do
    current 为当前执行进程
    ……
    将进程 current 加入到 lock 的阻塞链表中
    current. lock_blocked:=lock
    current. lock_state:=lock_type
    ……
    schedule()
done
```

在强二元信号量的基础上，可以实现 PIP 或 PCP 等协议。关于 PIP 和 PCP 的实现，限于篇幅，这里不作讨论。

**小结**　本文给出了改进 Linux 可抢占内核设计的基本思想，并主要针对互斥锁的改进支持实现提出了一个解决方案。目前，作为红旗 Linux 公司的实时产品 RFRTOS 采用了这一项改进方案，系统运行稳定，且在实时性能方面有较为显著的提高。需要指出的是，基于信号量互斥锁的设计提高了系统响应，但同时由于开放了更多并发线索，可能造成诸如系统死锁等的问题。增加系统的稳定性将是我们进一步的工作，其中包括增加检查死锁的机制等。

## 参 考 文 献


1 Dissecting DIAPM RTHAL-RTAI (with some comparisons to NMT-RTL, here and there) (draft), Paolo Mantegazza Dipartimento di Ingegneria Aerospaziale, Politecnico di Milano. http://www. aero. polimi. it/projects/rtai/

2 Barabanov, Yodaiken. Real-Time Linux . Linux Journal, Mar. 1996

3 Hill R, Srinivasan B, Pather D S. Niehaus Temporal Resolution andReal-Time Extension to Linux: [Technical Report ITTC-FY98-TR-11510-03]. Information and Telecommunication Technology Center. Electrical Engineering and Computer Science Departmaent, University of Kansas, June 1998

4 Ishiwata Y, ART Linux. http://www. etl. go. jp/etl/robotics/Projects/ART-Linux/, Jan. 2001

5 Tanenbaum A S, Woodhull A S. Operation Systems Design and Implementation, Second Edition. Prentice-Hall, 1997. 147～148

6 Lamport L. The Mutual Exclusion Problem. Joural of the ACM, April 1986

7 Stalling W. 操作系统内核与设计原理，第四版. 电子工业出版社，2001. 159～173

8 Sha L, Rajkumar R, Lehoczky J P. Priority Inheritance Protocols: An Approach to Real-Time Synchronization. IEEE Transactions On Computers, Sep. 1990

9 IA-32 Intel Architecture. Software Developer's Manual Volume 3: System Programming Guide, 2001