非独占锁的优先级继承协议及其在 Linux 下的实现

赵慧斌, 李小群, 孙玉芳
(中国科学院软件所一部, 北京 100080)

摘 要: 对于系统中的同步和互斥算法中, 支持优先级继承协议的互斥锁在提高实时系统调度精度, 增强系统的行为可预见性方面起到较为关键的作用. 但较早的优先级继承算法和实现并未考虑到非独占锁, 如读锁的优先级继承的问题, 本文提出一种基于读锁的优先级继承协议, 并给出了关于该协议的分析, 在此基础上, 基于 Linux 实现了该算法.

关键词: 实时; 非独占互斥锁; 优先级继承; 优先级反转; Linux; 可抢占核心
中图分类号: TP316.8 文献标识码: A 文章编号: 0372-2112 (2003) 08-1145-05

Priority Inheritance Protocol of Non-Exclusive Mutex and Its Implementation Under Linux

ZHAO Hui-bin, LI Xiao-qun, SUN Yu-fang
(Institute of Software, Chinese Academy of Sciences, Beijing 100080, China)

Abstract: Priority inheritance protocol is used to improve the scheduling precision of real time system with synchronization and exclusive requirements and hence increase the predictability of the behavior of an operating system. But the protocol and correspondent implementations do not involve non-exclusive mutex, such as reading lock. The paper gives rise to an algorithm of priority inheritance of non-exclusive mutex and analyzes the property of the protocol, on the basis of which an implementation under Linux is described.

Key words: real time; non-exclusive mutex; priority inheritance protocol; priority inheritance; linux; preemptable kernel

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 RTA[1], RT Linux[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 提出的中断虚拟层的方法[1], 减少了由于中断屏蔽而引进的调度延迟, 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. 硬实时化方案可以达到微秒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(服务质量) 保证, 又要求全面的操作系统服务. 尽管 KURT 通过改善 Linux 的时钟系统和增加调度模块的方式, 基本实现了这一目标, 但是, 由于在核心可抢占性方面并无本质提高, 限制了其应用的实时性, 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. 另外, 核心服务的细化仅仅增加了可抢占的粒度, 并未根本实现完全可抢占核心.

我们在开发红旗实时操作系统 RFRTOS 的过程中, 希望在不损失系统服务的情况下, 增加 Linux 的调度粒度和准确性, 从而改善其实时性, 而此时核心的可抢占性就成为决定实时性能较为重要的一个条件. Linux 欠缺核心可强占性设计方面的考虑, 体现在互斥锁设计方面有以下两方面的缺陷:

(1) 互斥锁设计较为简单, 难于保证互斥效率. Linux 的互斥锁较为普遍的基于 test and set[8] 指令集, 这种互斥锁机制实现简单、适用范围广, 但存在忙等待消耗处理器时间、不能防止饿死(starvation) 和易死锁等问题, 且不支持互斥锁协议;

(2) 优先级反转问题. 优先级反转[5] 是在指较低优先级持有互斥锁的过程中, 阻塞了欲获得同一互斥锁的高优先级进程的现象, 优先级反转破坏了可抢占核心的调度准则, 在实时操作系统中将带来不可预知的调度延迟.

收稿日期: 2002-09-12; 修回日期: 2003-02-24

对于互斥锁机制的改进,可采用基于信号量的机制实现互斥.基于信号量的互斥锁在处理器效率和防饿死方面较基于test and set 指令集的互斥锁为好,且可以支持多种互斥协议.而对于优先级反转现象,优先级继承协议(Priority Inheritance Protocol, PIP)[5]可以解决这一问题.PIP 协议可以保证较高优先级进程的阻塞时间在一个预知的范围之内,这样,就为调度分析提供了依据.

但文献[5]集中于讨论独占锁的优先级继承问题,而未涉及到非独占锁[6,7],如读锁[8]的优先级继承问题.而且并未给出参考实现方法,在实现方法上存在某些歧义.

ART2Linux 在改进 Linux 互斥锁机制的基础上实现了独占锁,如写锁的 PIP 协议.但 Linux 核心中提供了读锁的机制,而 ART2Linux 所采用互斥锁改进及 PIP 的实现回避了读锁机制,这显然将降低互斥效率.

本文就非独占锁的 PIP 协议,并提出了读锁的优先级继承协议,在此基础之上,给出了该算法的一些特性.同时,在我们开发的基于 Linux 的实时操作系统))) 红旗实时操作系统 RFRTOS 的过程中,实现了该算法.

本文首先给出与本文相关的符号和定义,随后针对读锁详细讨论读锁的优先级继承协议的算法、性质,最后给出在 Linux 下该算法的一种参考实现方法.

2 符号和定义

符号:

p 记 J_i 为一个作业,它是任务 S_i 的一个实例,P_i 和 T_i 分别表示任务 S_i 的优先级和周期,作业的优先级值为 i,优先级值 i 越小,作业优先级 P_i 越高;

p 记一个二元信号量锁为 S_i,P(S_i)和 V(S_i)分别表示对 S_i 的原子的上锁和解锁操作;

p 记作业 J_i 的第j 个关键区间为 $z_{i,j}$,也即作业 J_i 的第j 个 P,V 操作;记 $S_{i,j}$ 为保护 $Z_{i,j}$ 的的互斥锁.

p 记 $z_{i,j} < z_{i,k}$,如果关键区间 $z_{i,j}$ 完全包含于 $z_{i,k}$;

同时假定,所有关键区间之间的嵌套是合理的,即对给定的关键区间,或者 $z_{i,j} < z_{i,k}$,或者 $z_{i,k} < z_{i,j}$,或者 $z_{i,j} \mathrm{H} z_{i,k}$=<.另外,我们假定在一个关键区间内,同一互斥锁只能锁定一次;

p 记 $B_{i,j}$ 为所有较低优先级任务 J_i 可能阻塞 J_i 的所有关键区间的集合,即 $B_{i,j} = \{z_{i,k} | j > i$ 且 $z_{i,j}$ 可能阻塞 $J_i\}$;

p 由于仅考虑关键区间合理嵌套的情况,则关键区间的进入遵循一定的顺序.这样,可以将问题简化为考虑具有最大元素的 $B_{i,j}$ 集合,记为 $B^*_{i,j}$,也即 $B^*_{i,j} = \{z_{i,k} | (z_{i,k} \mathrm{I}\ B_{i,j}) \mathrm{C} (\sim \mathrm{v}\ z_{i,m} \mathrm{I}\ B_{i,j}$,且 $z_{i,k} < z_{i,m})\}$;

进一步,由于集合 $B^*_{i,j}$ 辟除了 $B_{i,j}$ 中关键区间包含的元素,可以将问题集中于 $B^* = Y_{j>i} B^*_{i,j}$ 的集合中,即包含了可能阻塞作业 J_i 的所有最长关键区间的集合.

定义:

p 定义作业 J 被作业 J_i 的第j 关键区间 $z_{i,j}$ 阻塞,如果作业 J_i 较作业 J 的优先级为低,但作业 J 必须等待作业 J_i 退出关键区间 $z_{i,j}$,才可以继续执行;

p 定义作业 J 通过互斥锁 S 被作业 J_i 阻塞,如果关键区间 $z_{i,j}$ 阻塞了作业 J 的执行,且 $S_{i,j} = S$.

我们同时假定系统是完全的可抢占的,即在任意时刻,系统所执行的进程是所有可执行进程中,优先级最高的进程.

3 读锁模型

与写锁的独占性相反,读锁可同时允许多个读线程同时进入,同时进入的线程数量由读锁的最大允许线程数限定,一旦同时获取读锁的线程数达到读锁的最大允许线程数,则欲获得读锁的其他线程将被阻塞,直到某个获得读锁的线程放弃该读锁为止.

这里给出一个基本的读锁实现的功能代码描述,如图 1 所示.

```
struct semaphore {
    int reader_numbers;
    QueueType BlockedQueue;
}
void read_lock(semaphore s) {
    s.reader_numbers++;
    if(s.reader_numbers == 最大进程数)
    {
        将当前进程加入 s.BlockedQueue;
        阻塞该进程;
    }
}
void read_unlock(semaphore s) {
    s.reader_numbers--;
    if(s.BlockedQueueX<) {
        从 s.BlockedQueue 将进程 P 移出;
        s.reader_number++;
        将时程 P 设为可执行状态;
    }
}
```

图 1 读锁的功能代码

为简化问题,文中如未加说明,每个读锁所允许的最大线程数均相同.

4 读锁的优先级继承协议定义及其性质

4.1 读锁的优先级继承协议定义

定义一个受读信号量锁保护的关键区间,该关键区间将含有一个读进程计数器,当读作业获得互斥锁,进入关键区间,则该关键区间的读计数器加 1,如果读计数器等于关键区间允许的最大读作业数,则读作业将被阻塞.

p 作业 J 是所有等待执行作业中最高优先级的进程,它将获得处理器.在作业 J 进入一个关键区间之前,它必须首先获得保护关键区间的信号量 S.但如果信号量 S 此前已被上锁,且读作业的计数等于关键区间所允许的数量 N,则作业 J 将被阻塞,对信号量 S 的操作将被禁止.在这种情况下,我们称作业 J 被持有信号量互斥锁 S 的作业阻塞.否则,作业 J 将获得信号量互斥锁 S,进入关键区间,关键区间的读作业计

数加一. 当作业退出关键区间, 且关键区间的读作业不为 0, 关键区间的读作业计数减一, 此时如果存在阻塞的作业, 则其中最高优先级的作业将被唤醒. 否则, 保护关键区间的信号量锁将被释放;

p 作业 J 使用预设的优先级, 直到其进入关键区间, 且关键区间的读作业计数等于关键区间所允许的读作业数 N, 如果因此阻塞了更高优先级的作业. 此时, 作业 J 将继承被阻塞的最高优先级的进程, 而其他获得互斥锁 S 的作业如果优先级低于被阻塞的作业, 也将继承被阻塞的作业的优先级. 当作业 J 或任意一个获得互斥锁 S 的作业退出该关键区间, 发生优先级继承的作业将恢复到其进入该关键区间之前的优先级;

p 读作业的优先级继承是具有传递性的. 这与基本的优先级继承协议是一致的.

412　读锁的优先级继承的性质

在分析读锁的优先级继承协议的性质, 为简化问题, 我们假定系统不发生死锁现象, 比如, 互斥锁的申请和放弃按照特定的顺序.

引理 1　作业 J_H 可能被较低优先级的作业集合 $J_{L1}, J_{L2}, \cdots, J_{LN}$ 阻塞, 则阻塞仅可能在以下条件下发生: 作业 J_H 初始执行时, 集合 $J_{L1}, J_{L2}, \cdots, J_{LN}$ 已处于读锁保护的关键区间 $z_{L,k} \in B^*_{H,L}$.

证明　该引理的充分性: 作业集合处于读锁保护的关键区间, 则它们或者直接阻塞 J_H, 或者因为间接的优先级继承, 优先级提升而获得高于 J_H 执行的权限;

该定理的必要性: 用反证法, 假设作业集 $J_{L1}, J_{L2}, \cdots, J_{LN}$ 阻塞了 J_H 的执行, 并且其中至少一个作业 J 不处于任何关键区间 $z_{L,k} \in B^*_{H,L}$, 则该作业 J 必定优先级小于 J_H, 这样就出现了低优先级任务 J 先于 J_H 执行的情况, 这与我们的系统可抢占假设矛盾. 由此, 定理的必要性得到证明.

引理 2　如果采用读锁的优先级继承协议, 则较高优先级的作业 J_H 最多被较低优先级作业集 $J_{L1}, J_{L2}, \cdots, J_{LN}$ 阻塞集合 $B^*_{H,L}$ 中的一个关键区间, 不论 J_H 和 $J_{L1}, J_{L2}, \cdots, J_{LN}$ 共享多少个互斥锁.

证明　根据引理 1, $J_{L1}, J_{L2}, \cdots, J_{LN}$ 仅在同处于关键区间 $z_{L,k} \in B^*_{H,L}$ 时, 才能阻塞 J_H 的执行, 则一旦作业集 $J_{L1}, J_{L2}, \cdots, J_{LN}$ 任一作业放弃 $z_{L,k}$, 则 J_H 抢占执行, J_H 也就不可能被作业集中其他作业阻塞.

定理 1　在读锁的优先级继承之下, 较高优先集作业 J_0 和 n 个作业集合 $\{JSet_1, \cdots, JSet_n\}$, 其中, $\{JSet_i \mid J_{Li1}, J_{Li2}, \cdots, J_{LN}$ 均处于 $z_{i,k}$, 且 $z_{i,k} \in B^*_{0,i}\}$, 即引理 2 中的作业集; 作业 J_0 最多被每个集合 $B^*_{0,k}, 1 \leqslant k \leqslant n$ 中一个关键区间所阻塞.

证明　由引理 2, J_0 最多被作业集 $JSet_i$ 阻塞 $B^*_{0,i}$ 中的一个关键区间, 对 n 个作业集合, 则最多被每个 $B^*_{0,k}, 1 \leqslant k \leqslant n$ 集合中的一个关键间所阻塞.

推论 1　在读锁的优先级继承之下, 较高优先级作业 J_0 和 n 个较低作业 $J_1, \cdots, J_N$, J_0 在其执行过程中使用 m 个互斥锁, 则 J_0 最多被 $B_p^{max}(m, n, N) = \begin{cases} 0, & \text{如果 } n < N \\ m, & \text{如果 } n \geqslant N \end{cases}$ 个集合 $B^*_{0,k}, 1 \leqslant k \leqslant n$ 的一个关键区间所阻塞, 式中, N 为读锁保护的关键区间所允许同时进入的读作业数量.

证明　我们将 n 个作业按这样的顺序分组: 如果其中 N 个作业组成了作业集合 $\{JSet \mid J_{L1}, J_{L2}, \cdots, J_{L,N}, J_{L1}, J_{L2}, \cdots, J_{LN}$ 同处于 $z_{i,k} \in B^*_{0,L}\}$, 则分为一组, 其中, N 为读锁保护的关键区间所允许同时进入的读作业数量; 因为一个较低优先级作业 $J_L, 1 \leqslant L \leqslant n$ 可以同时持有多个不同的互斥锁, 则如果作业 J_0 与 J_L 共享所有的 m 个互斥锁, 且 J_L 在某一时刻同时持有这 m 互斥锁, 更进一步, 所有的较低优先级作业与 J_L 情况相同, 则有以下两种情况:

(1) $n < N$, 则同处于一个读锁的进程数目不足以构成阻塞, 因而不会阻塞 J_0;

(2) $n \geqslant N$, 则最多形成 m 个定理 1 所示的阻塞集合, 根据定理 1, J_0 最多被阻塞 m 个集合 $B^*_{0,k}, 1 \leqslant k \leqslant n$ 中的一个关键区间.　证毕

文献[1]证明:

定理 2　在基本的优先级继承协议之下, 如果有 m 个互斥锁可以阻塞作业 J, 则作业 J 最多可被阻塞 m 次. 该结论同样适用于非独占互斥锁的优先级继承协议.

根据定理 1 和定理 2, 可以得到: 在非独占互斥锁优先级继承协议, 作业 J 最多可以被阻塞 $\min(m, B_p^{max})$ 个关键区间. 其中 m 为可能阻塞作业 J 的互斥锁数目, B_p^{max} 为推论 1 中的阻塞区间数目.

5　在 RFRTOS 中的实现

由于读锁的优先级继承协议是基于信号量的互斥锁. 虽然 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;

(2) 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;

(3) 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. 我们实现的支持优先级继承协议的互斥锁, 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.

关于宏内核的设计问题, 我们采用了 ART2Linux 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, 在改进的互斥锁基础之上, 我们进一步实现了针对实时任务的优先级继承协议, 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.

511　数据结构

如图 2(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, 在 RFRTOS 设定可以同时进入

关键区间的最大读作业数目为 16 个. 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_struct* prochead, ,
    int proto;
    mutex_list_t
        owners [允许的最大读进程数];
}
struct mutex_list_t {
    struct task_struct* proc;
    struct mutex_list_t* prev;
, ,
}
```

图 2(a) mutex_t 的数据结构

```
struct task_stuct {
    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 2(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 2(b)所示, 增加 rt_base_priority 成员是进程未发生优先级继承的优先级, 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.

5.1.2 算法

读锁的申请是由 read_lock 算法完成的, 该算法依据第 3 节中的算法模型, 在此不再赘述.

read_lock 算法:

```
参数: lock is type of mutex_t
do
    current is currently running process
    3 if lock.state X LOCK_WRITE
        AND
      (lock.proto 不是优先级继承协议)
        OR
      lock.num_readers X 最大读进程数)
        AND
      (lock.proheadX <
        OR
      lock.prochead.rt_priority3 current.rt_priority)
    then
      将 lock 加入到 current.locks_held 列表
      lock.num_readers+ +
      lock.state: = LOCK_READ
    else
      do_locking(lock, LOCK_READ)
    fi4
  done
```

进程如果发生阻塞, 将进入 do_locking 流程, 该流程涉及到优先级继承算法, 与此相对的是解锁时的优先级恢复算法. 可以看到, 在提供优先级继承协议情况下, 阻塞流程将进入优先级继承的处理流程, 在 RFRTOS 中, 优先级继承依赖于 inherit_prio, 算法描述如下:

do_locking 算法:

```
参数: lock is type of struct mutex_t, lock_type is 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
    if lock.proto 为优先级继承协议 then
        inherit_prio(current)
    fi
    , ,
    schedule( )
done
```

inherit_prio 算法:

```
参数: prock is type of task_struct)
do
  mtx is type of mutex_t
  mtx= proc.lock_blocked
  priority is type of int
  priority: = proc.rt_priority;
  if mtx.state= LOCK_READ then
    for 每个持有 mtxcs 的进程 P
      即 P p, pI mtx.owner[0..CONFIG_
          MAX_READERS- 1].proc
      do
        if P.rt_priority3 priority then
          P.rt_priotyity: = priority
          if P 被阻塞 then
            根据进程 P 的优先级变化, 调整
            进程 P 的互斥锁阻塞队列
            inherit_prio(P)
          else
            将进程 P 标志为可执行
          fi
```

```
        fi
    done
  fi
done
```

与 read_lock 和 inherit_prio 相对应的是 read_unlock 和 recal_prio, 分别完成解除读锁和恢复解锁后的进程优先级, 由于篇幅所限, 不再赘述.

6 小结

本文提出了基于读锁的优先级继承协议算法, 并分析证明了该算法的性质, 在此基础之上, 给出了该算法在红旗实时操作系统 RFRTOS 上的一种参考实现. 由于现代操作系统在互斥锁实现方面的进一步细分, 非独占锁, 如读锁获得了更为广泛的应用, 因而本文提出的基于读锁的优先级继承协议算法将为进一步的实时操作系统的设计及调度分析提供依据.

参考文献:

[1] Paolo Mantegazza. Dissecting DIAPM RTHAL2RTAI (with some com2 parisons to NMI2RTL, here and there) (draft) [DB/OL]. http://www.aero.polimi.it/projects/rtai/, 2000211.

[2] Michael Barabanov. A Linux2based Real2Time Operating System [D]. New Mexico: New Mexico Institute of Mining and Technology, Socorro, June 1997.

[3] R Hill, B Srinivasan, S Pather, D Niehaus. Temporal Resolution and Real2Time Extension to Linux, Technical Report ITTC2FY982TR2 11510203 [R]. Information and Telecommunication Technology Center, Electrical Engineering and Computer Science Department, University of Kansas, June 1998.

[4] Youichi Ishiwata. ART Linux [DB/OL]. http://www.etl.go.jp/etl/robotics/Projects/ART2Linux/, 2001201205.

[5] Sha L, Rajkumar, R, Lehoczky J P. Priority Inheritance Protocols: An Approach to Real2Time Synchronization [C]. IEEE Trans, September 1990.

[6] Andrew S Tanenbaum, Albert S Woodhull. Operation Systems Design and Implementation, Second Edition [M]. New Jersey: Prentice2Hall, 1997. 47- 148.

[7] Lamport L. The mutual exclusion problem [J]. Journal of the ACM, ACM Press, April 1986.

[8] William Stalling. 操作系统内核与设计原理, 第四版 [M]. 北京: 电子工业出版社, 2001. 159- 173.

作者简介:

赵慧斌　男, 1974 年出生于内蒙古, 博士生, 现就读于中科院软件所, 主要从事实时操作系统的研究以及基于 Linux 的嵌入式实时系统开发工作.

李小群　女, 1969 年 6 月 13 日出生于陕西, 博士, 主要从事实时操作系统的研究以及基于 Linux 的嵌入式实时系统开发工作.

孙玉芳　男, 1947 年 2 月出生于江苏, 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.