# Linux 2.6 内核调度机制剖析与改进①

## Analysis and Improvement of the Process Scheduling Mechanism in Linux Kernel 2.6

张永选　(江西经济管理干部学院 电子计算机应用系 江西 南昌 330088)

毛红梅　(南昌航空大学 计算机学院 江西 南昌 330063)

**摘　要：** 详细剖析了 Linux 2.6 内核进程调度的时机、调度策略、调度有关的重要数据结构、O(1)调度算法实现，以及 Linux 2.6 内核新引入的内核抢占机制。为了使 Linux 2.6 内核支持硬实时应用，提出了改进的最小裕度优先调度算法。该算法通过引入抢占阈值，从而减少颠簸现象造成的系统资源浪费，并提高了实时处理效率。

**关键词：** 内核 进程调度 O(1) LSF

进程调度系统是操作系统的灵魂，其主体——进程调度程序是操作系统中最为核心的部分之一[1]。同时也是任何多任务操作系统中执行频度最高的部分之一，其性能的好坏对操作系统整体性能有着至关重要的影响。

Linux2.6 采用了一种全新的调度算法。此调度算法具有如下特点：(1)完全实现 O(1)算法。不管有多少进程，调度程序选择进程时所需的时间是恒定的；(2)增强了 SMP 的可扩展性。每个处理器拥有独立的锁和可运行队列，不会产生互斥；(3)强化 SMP 的亲和力。尽量将相关一组进程分配到一个 CPU 上连续地执行，只有在需要平衡时才会进行进程迁移，不会频繁地产生进程跳跃；(4)增强了对软实时性的支持。提高了交互性能，保证系统及时相应；(5)保证了公平性。在合理的时间范围内没有进程会处于饥饿状态，也没有进程会得到大量的时间片。但是 Linux 操作系统本身是按照通用分时操作系统的思路设计的，使其不适用于实时系统中。

基于上述分析，本文提出了改进的最小裕度优先调度算法，此算法在传统最小裕度优先调度算法的基础上，通过引入抢占阈值以达到减少颠簸，提高系统效率的目的。

① 基金项目:2006 年度江西省自然科学基金(0611092)
收稿时间:2009-02-23

## 1　Linux 2.6 内核调度机制的剖析

### 1.1 响应时间

表 1 显示了 Linux 2.6.15 内核在平均情况下的响应时间。测试使用的是带有超过 5 个中断的 Lynux Works 实时测试设备，该设备使用一颗 PⅢ 1.0GHz 处理器。系统处在由持续地磁盘数据传送、网络通信、控制台输入、图像处理和一个定时卡组成的高负载下运行，共采集了 310 万个样本。

表 1　Linux 2.6.15 内核响应时间

| 中断响应(毫秒) | 任务响应(毫秒) |
|---|---|
| 14 | 132 |

可见，从通用分时操作系统的标准看，Linux 具有良好的实时性；但对于实时操作系统来说，显然还有很大差距。

### 1.2 内核的调度时机

进程的调度时机与引起进程调度的原因和进程调度的方式有关。在 2.6 内核中，除核心应用主动调用调度器之外，核心还在应用不完全感知的情况下在以下三种时机中启动调度器工作：

(1) 从中断或系统调用返回到用户态；

(2) 某个进程允许被抢占 CPU;

(3) 进程主动进入休眠状态。

### 1.3 内核的调度策略

Linux2.6 内核中进程的调度策略有以下几种:

(1) SCHED_FIFO 先进先出式调度,除非有更高优先级进程申请运行,否则当前进程将保持运行至退出才让出 CPU;

(2) SCHED_RR 轮转式调度,当前进程被调度下来后将被置于运行队列的末尾,以保证其它实时进程有机会运行;

(3) SCHED_NORMAL? 常规的分时调度策略。

SCHED_NORMAL 是普通进程的调度策略;SCHED_FIFO 和 SCHED_RR 都是实时进程的调度策略。O(1)调度器是以进程的动态优先级 prio 为调度依据的,它总是选择目前就绪队列中优先级最高的进程作为候选进程 next。由于实时进程的优先级总是比普通进程的优先级高,故能保证实时进程总是比普通进程先被调度。

2.6 内核中,优先级 prio 的计算不再集中在调度器选择 next 进程时,而是分散在进程状态改变的任何时候。并且根据计算结果调整其在就绪队列中的位置。

## 2 内核调度有关的数据结构

### 2.1 进程优先级的划分

Linux2.6 内核中将进程优先级作了以下规定:进程优先级范围是从 0~MAX_PRIO-1,其中实时进程的优先级的范围是 0~MAX_RT_PRIO-1,普通进程的优先级是 MAX_RT_PRIO~MAX_PRIO-1。数值越小优先级越高。

### 2.2 就绪队列 struct runqueue

structrunqueue 是 2.6 内核调度器中一个非常重要的数据结构,它主要用于存放每个 CPU 的就绪队列信息。限于篇幅,这里只介绍其中最为重要的部分:

(1) prio_array_t?*active,?*expired,?arrays[2]

这是 runqueue 中最重要的部分。每个 CPU 的就绪队列都是一个数组,按照时间片是否用完将就绪队列分为两个部分:活动数组和过期数组,分别用指针 active 和 expired 来指向数组的两个下标。

当一个普通进程的时间片用完以后将重新计算进程的时间片和优先级,并将该进程从 active array 移到 expired?array 中相应优先级的进程队列中。当 active?array 中没有进程时,则将 active 和 expired 指针调换一下就完成了切换工作。这是 2.6 内核调度器一个重要特点。

(2) spinlock_t lock

就绪队列 runqueue 的自旋锁。当对 runqueue 进行操作的时候,需要对其加锁。

### 2.3 进程标识 task_struct

Linux 内核使用 task_struct 结构来表示进程。由于 task_struct 结构体比较复杂,我们只涉及它与进程调度相关的重要部分。

(1) int prio,static_prio

进程的动态优先级和静态优先级。prio 表示进程的动态优先级,是调度的唯一依据。static_prio 表示进程的静态优先级。一个进程的初始时间片的大小完全取决于它的静态优先级。

(2) unsigned long sleep_avg

进程的平均等待时间。sleep_avg 反映了该进程需要运行的紧迫性。当进程休眠时该值增加,进程运行时该值减少。它是影响进程优先级最重要元素。值越大,说明该进程越需要被调度。

## 3 Linux 2.6内核调度算法分析

### 3.1 schedule()代码分析

schedule()函数是实现进程调度的主要函数,并完成进程切换工作。schedule()用于确定最高优先级进程的代码非常快捷高效。它在/kernel/sched.c 中的定义如下:

```
…
task_t *prev, *next;
runqueue_t *rq;
prio_array_t *array;
int idx;
…
preempt_disable();
prev = current;
```

2.6 内核支持抢占,所以在对队列操作时需要设置为不可抢占。

```
rq = this_rq();
…
array = rq->active;
if (unlikely(!array->nr_active)) {
```

```
    rq->active = rq->expired;
    rq->expired = array;
    array = rq->active;
    rq->expired_timestamp = 0;
    rq->best_expired_prio = MAX_PRIO;
}
```

这段代码的作用是执行两个数组（活动数组 active 和过期数组 expired）的切换。首先，如果活动数组中没有进程了，则通过指针操作来切换两个数组。之前我们看到在过期数组中的进程时间片已经被计算好了。所以在两个数组切换后，过期数组中的进程都变为活动进程，交换数组的时间就是交换指针的时间。这种交换就是 O(1)调度算法的核心。整个 O(1)算法的工作机制可以用图 1 形象地描述：

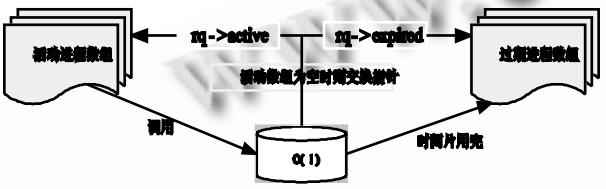


图 1　O(1)算法的工作机制

既然已经有了活动数组，并且各个进程都按优先级排好队等待被调度，接下来就要选择候选进程 next 了。接着看代码：

```
idx = sched_find_first_bit(array->bitmap);
queue = array->queue + idx;
next = list_entry(queue->next, task_t, run_list);
…
    if (unlikely(next->prio != new_prio)) {
        dequeue_task(next, array);
        next->prio = new_prio;
        enqueue_task(next, array);
    } else
        requeue_task(next, array);
…
```

首先要在活动数组中的进程队列里找到第一个被设置的优先级位，这里通过 sched_find_first_bit 函数来实现。找到第一个被设置的位以后，接着找到与该位相应的优先级的对应可运行进程队列，然后再找到该队列中排列的第一个进程(由 next 指向该进程)；最后把找到的候选进程 next 插入运行列表中。整个 O(1)调度算法找到候选进程的过程大致如图 2 所示：

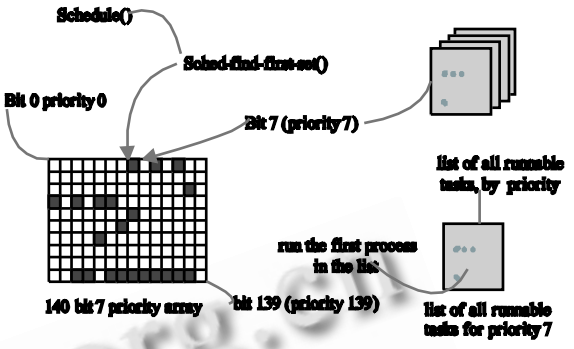


图 2　O(1)调度算法找到候选进程的过程

图 2 中的网格为 140 位优先级数组，queue[7]为 active 数组中优先级为 7 的就绪进程列表。

```
…
if (likely(prev != next)) {
…
        prev = context_switch(rq, prev, next);
…
} else
        spin_unlock_irq(&rq->lock);
…
```

如果候选进程 next 不是当前运行进程，则需要进行进程切换。反之则无需切换，仅仅释放之前对运行队列所加的锁即可。整个 schedule()函数的运行流程如图 3 所示：

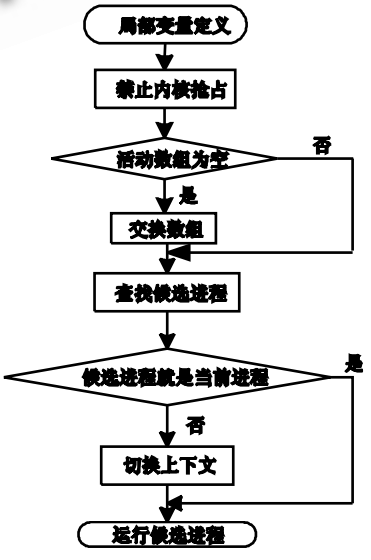


图 3　schedule()函数的运行流程

### 3.2 内核抢占

2.4 版之前的 Linux 和大部分 Unix-Like 操作系统一样，只支持用户抢占，不支持内核抢占[2]。为了提高对交互式进程的支持并更好地实现软实时工作方式，2.6 内核引入了内核抢占。现在，只要重新调度是安全的，那么内核就可以在任何时间抢占正在执行的进程[2]。

只要进程没有持锁，内核就可以进行抢占。为了支持内核抢占，2.6 内核对每个进程的 thread_info 结构体(进程描述符 task_struct 的成员变量，用来保存内核需要经常访问的重要信息)引入了成员变量 preempt_count 计数器。该计数器初始值为 0，每当使用锁的时候数值加 1，释放锁的时候减 1。当数值为 0 时，内核就可以执行抢占。

## 4 一种改进的实时调度算法

为了使 Linux 2.6 内核支持硬实时应用环境，笔者给出了一种改进的实时调度算法——改进的最小裕度优先(LSF)调度算法。

### 4.1 改进的最小裕度优先调度算法

最小裕度优先(Least Slack First)调度算法是实时系统中较为常见的动态优先级调度算法，它是对最早截止期优先(EDF)算法的改进[3]。这种算法依据富裕时间来确定任务的优先级，该富裕时间即为“裕度”。裕度小的，优先级高，反之则低。在该算法中，通过结合任务执行的缓急程度来给任务分配优先级。它克服了 EDF 算法中因仅仅考虑截止期限而带来的缺陷。在 LSF 算法中，裕度 S 定义如下：

$$S = (d-t) - c'$$

其中，d 为任务的截止期限，t 为系统当前时间，c'为 t 时刻任务的剩余部分的执行时间。在裕度相同时，截止期限靠前的任务的优先级高；当 $s \geqslant 0$ 时任务才会被调度，否则就会因来不及处理而夭折。但是这种算法存在一个问题：当系统中有 1 个以上(假设有两个，分别为 $T_1$、$T_2$)最小裕度相近或相等的任务，且系统中没有比这两个任务裕度更小的任务了。由于正在执行的任务(假设为 $T_1$)的裕度不变，而在就绪队列上等待的任务($T_2$)的裕度则随着时间的推移而减小，从而使它们的优先级动态地发生变化。因此，随着调度的执行，$T_2$ 就会因裕度变得足够小而抢占 CPU。经过任务切换后，同样的道理，过了一段时间，$T_1$ 也会抢占 CPU，如此反复进行。这种频繁的任务切换即为颠簸现象(Thrashing)[4]。

为了减少颠簸现象造成的系统资源浪费，对 LSF 算法进行如下改进：每个任务除了按裕度分配优先级外，还有一个抢占阈值，从而构成一个双优先级系统。一旦任务得到 CPU，其优先级就提升到其抢占阈值的水平，直到它的执行结束或被其它任务抢占后再回复到其原先的优先级。设定任务的优先级值越大，其优先级越高，任务的抢占阈值大于其优先级值。

通过引入抢占阈值，可以有效地减少颠簸现象的发生。现在还是假设有两个任务，分别为 $T_1$、$T_2$，它们的最小裕度相近或相等。$T_1$ 的裕度(优先级值)和抢占阈值分别为 $S_1$ ,$P_1$，$T_2$ 的裕度和抢占 $S_2$，$P_2$。开始 $T_1$ 正在执行，$T_2$ 则等待。随着时间的推移，$S_2$ 不断增大，$S_1$ 则保持不变。当 $S_2 > S_1$ 时,并不进行抢占,而是让 $T_1$ 继续运行。直到 $S_2 > P_1$ 时，$T_2$ 才可以抢占 $T_1$。于是，在前述两个任务的情况下，改进后的调度算法就减少了颠簸现象。

这种基于抢占阈值的调度算法包含了完全抢占和非抢占的特点，属于有条件的抢占调度模型[5]。当每个任务的抢占阈值与其优先级相同时，模型就还原成完全抢占的 LSF 算法；当每个任务的抢占阈值都充分大时，就变成了非抢占调度模型。

至于抢占阈值的设定，它是改进算法的关键。直接影响到任务切换的频率，也影响到任务截止期的错失率，还影响到 CPU 的有效利用率。根据前面有关优先级值越大，优先级越高的假设，任务的抢占阈值需要设定成不小于相应的优先级。如果取抢占阈值大于最大优先级值，则调度算法退化为非抢占模型。因此，在这里，抢占阈值的取值范围为大于进程优先级，小于当前所有进程最大优先级值。

### 4.2 改进的 LSF 算法在 Linux 中的实现

为了在 Linux 中实现上述改进的 LSF 算法，需要进行的主要改进如下：

(1) 在进程控制块 task_struct 中增加“裕度”相关的属性：任务剩余执行时间 Ctime(初值为估计运行时间)、任务提交时间 Etime、任务相对截止期限 Dtime、裕度值 Stime。设 T 为系统当前时间，则由裕度的定义公式可得：

$$Stime = (Etime + Dtime) - T - Ctime$$

(2) 保持 2.6 内核的运行队列结构以及候选任

务的选取方法，实时任务的优先级属性值一经初始设定后就不再改变。同一优先级队列中的实时任务按照其裕度值从小到大有序排列，而不是采用原来的先进先出形式。于是，新的任务调度依据变为：任务优先级越大，越优先得到调度。原系统中实现将进程插入到相应优先级队列末尾的函数 enqueue_task()需要修改，使其按裕度值的大小顺序进行插入操作。

(3) 时钟中断子程序 scheduler_tick()中增加对裕度值的实时更新处理。具体就是随着时钟的嘀嗒而增加运行进程的 Ctime 值，并减少等待进程的 Stime。此外，基于抢占阈值的候选任务的选取操作也要在时钟中断子程序中做相应的修改。

(4) 对于按照时间片轮转方式运行的实时任务，由于优先级队列的有序性，使得用完时间片的任务不能再通过插入到队尾的方式保留在 active 数组中了，而是需要插入到 expired 数组中。

(5) 候选任务选取阶段的改进最大。系统不再是简单地选取第一个就绪任务作为候选任务，而是要在此基础上进行必要的筛选：对于实时候选任务，由于引入了裕度和抢占阈值，只有其裕度值大于 0，方可作为合格的候选进程，然后将其优先级与当前任务的抢占阈值比较。如果候选任务的优先级大于当前任务的抢占阈值，则进行抢占，反之则不抢占[6]；对于非实时候选任务，只有无实时任务的情况下才会被调度。

根据以上分析，对 Linux 2.6.15 内核代码修改后重新编译，并在第 1.1 节所述的环境下重新测试其平均情况下的响应时间，得到的数据如表 2 所示。

从表中数据看出，Linux 内核的平均响应时间提高了大概 3 个数量级，达到了硬实时应用的要求。

表 2　改进后的 Linux 2.6.15 内核响应时间

| 中断响应(微秒) | 任务响应(微秒) |
|---|---|
| 8 | 13 |

## 5　结语

Linux2.6 版内核实现了一个高效的 O(1)调度器，具有更好的实时性能、重负载下更高的 CPU 使用率、以及交互作业更快的响应时间等优良特性。但 2.6 版内核的仍然不能满足硬实时应用的要求；可抢占内核也只限于对 CPU 的抢占，还不支持对内存等其他资源的抢占[7]。针对 2.6 版内核不支持硬实时应用环境的不足，本文给出了一种基于最小裕度优先(LSF)调度算法改进方案，并具体给出了在 Linux 下的实现方法。

## 参考文献

1 何克右,周彩贞.Linux2.6 进程调度机制的剖析.华中师范大学学报(自然科学版), 2007,(4):176－183.

2 Love R. Linux 内核设计与实现(第 2 版).Novel Press, 2005.

3 Terrasa A, Garcia-Fornes A, Botti VJ. Flexible real-time linux:A flexible hard real-time environment. Real-Time Systems, 2004,22(2):151－173.

4 陈莉君,张琼声,张宏伟.深入理解 LINUX 内核(第三版).北京:中国电力出版社, 2007.

5 金宏,王强,王宏安,等.基于动态抢占阈值的实时调度.计算机研究与发展, 2004,(3):393－398.

6 许占文,李歆.Linux 2.6 内核的实时调度的研究与改进.沈阳工业大学学报, 2006,(8):438－441.

7 丁聪,张玉璘. Linux 2.6 进程调度算法实时性能改进.济南大学学报(自然科学版), 2008,(4):56－63.