Linux 中一种改进的实时调度算法及其应用*)

谭云福　刘　杰　刘国华
（燕山大学信息科学与工程学院　秦皇岛 066004）

摘　要　在实时操作系统中，调度算法起着关键性的作用，然而调度算法的开销与系统的调度性能之间经常是一对矛盾。就此问题，结合最新版 Linux2. 6 内核任务调度的特点，提出了一种改进的最小裕度优先(LSF)算法。针对 LSF 算法中因任务间的频繁切换造成系统开销增大的缺点，通过采用适当的抢占阈值策略减少“颠簸”现象，提高了 Linux2. 6 内核的实时性。

关键词　Linux，实时调度，颠簸，抢占阈值，最小裕度优先

Improved Real-time Scheduling Algorithm and its Application in Linux

TAN Yun-fu　LIU Jie　LIU Guo-hua
(College of Information Science and Engineering, Yanshan University, Qinhuangdao 066004, China)

Abstract　In a real-time operating system, scheduling algorithm plays the vital role, but the cost of scheduling algorithm and the effect of scheduling in the system are often a pair of contradictions. Contacting the characteristics of the task scheduling in the newest edition of Linux2. 6 kernel, an improved algorithm of least slack first(LSF) on the Linux2. 6 is realized. To solve the problem of the costly system for frequently switching among the tasks in LSF, we use proper preemption threshold policy to reduce the case of thrashing, and the real-time performance in the Linux kernel is improved.

Keywords　Linux, Real-time scheduling, Thrashing, Preemption threshold, LSF

1　引言

目前，许多嵌入式实时应用系统采用了在 Linux 基础上改进的操作系统，除了通过提高 Linux 时钟精度、解决关中断和核心态不能被抢占的问题外，通过改善分配任务优先级的调度策略也可以从很大程度上改善系统的实时性能，真正的嵌入式实时操作系统往往都是基于优先级改进的[1,2]。

最小裕度优先(Least Slack First)调度算法是实时系统中较为常用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，根据任务所剩余的空闲时间(即裕度)的多少来分配优先级。裕度越少，说明该任务就越紧急，就越需要尽快执行，这样保证了紧急任务(而非截止期越早的任务)的优先执行。但由于等待任务的裕度是严格随时间递减的，在系统执行过程中，等待任务可能会由于紧急程度的提高而又抢占当前执行的任务，从而造成任务之间的频繁切换(又称为颠簸——thrashing)，使系统性能下降。利用抢占阈值的调度策略，通过控制不必要的任务抢占，可降低由于任务抢占引起的系统开销和过多的上下文切换[3]。但过于复杂的调度算法也会消耗太多的系统资源，反而会降低系统的调度性能。调度算法的开销与系统的调度性能之间经常是一对矛盾，从而产生了一个如何找到最合适的优先级和抢占阈值的问题。

目前，Linux 内核的最新版是 2. 6，从 2. 6 版开始引入了核心态可抢占、O(1)调度算法等新技术，有效地提高了系统的实时性。但对于实时性要求较高的应用场合，Linux2. 6 调度器仍然存在缺陷——没有实现实时任务的动态调度策略和算法，不能为实时任务计算动态优先级，容易使实时任务因得不到及时调度而被迫夭折，降低了系统的实时调度性能[4]。本文根据基于抢占阈值的 LSF 算法思想，并结合 Linux2. 6 内核调度器的特点，选择了一种简化的抢占阈值和计算优先级的方法，实现了动态实时调度，有效地增强了 Linux 内核的实时性。

2　算法的原理

2.1　LSF 调度算法及颠簸现象

LSF 算法是对最早截止期限优先(Earliest Deadline First)算法的改进。在 EDF 算法中，由于在确定各任务的优先级时只考虑了截止期限这一个因素，没有考虑任务执行时间的影响，因此有一个明显的缺陷，就是从任务获得处理器开始到任务的截止期限这段时间是否能让其执行完毕。如若不能，任务会因来不及处理而夭折。在 LSF 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。假设在 t 时刻，任务剩余部分的执行时间为 X，截止期限(绝对)为 D，则该任务的空闲时间(裕度)为

$$S=D-t-X \tag{1}$$

假设任务的优先级由裕度大小决定，裕度越小，优先级越高；裕度相同时，截止期限靠前的任务的优先级高，先执行；当 $S\geqslant 0$ 时任务才可以调度，否则被夭折。当系统中有 1 个以上最小裕度相近或相等的任务时，由于正在运行的任务(假设为 T_i)裕度 S_i 不变，而在就绪队列上等待的任务的裕度则随时间的推移而减小，从而使它们的优先权动态地发生变化。因此，随着调度的执行，总会有某个等待的任务(假设为 T_j)裕度变得足够小而抢占 CPU。经此任务切换后，T_j 成为当前

*)国家自然科学基金(编号 60773100)。**谭云福**　副教授，硕士生导师，研究方向为嵌入式系统与应用；**刘　杰**　硕士，研究方向为嵌入式 Linux 系统及应用；**刘国华**　教授，博士生导师，研究方向为软件工程、数据库理论及应用。

任务，而 T_i 成为新的等待任务。在 T_j 执行期间，其裕度 S_j 保持不变，等待任务 T_i 的裕度 S_i 随时间递减，当 $S_i<S_j$ 时，T_i 又抢占 T_j 重新获得执行，从而又发生一次任务切换。这样，T_i 和 T_j 会不断相互抢占执行。这种频繁的任务切换现象称为颠簸现象。如图 1 所示，设任务 $T0,T1,T2$ 的初始状态为：$S0(0)=9,X0(0)=3,D0=12;S1(0)=9,X1(0)=4,D1=13;S2(0)=9,X2(0)=5,D2=14$。它们在调度过程中产生了颠簸现象。

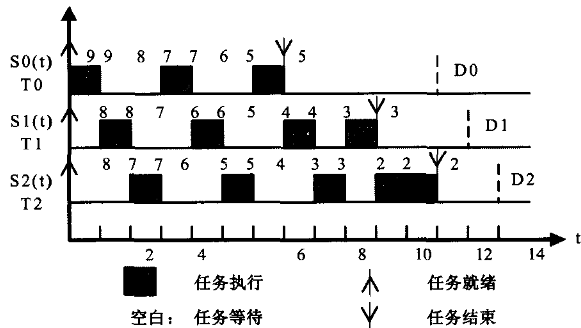

图 1　LSF 算法中的颠簸现象

在两个以上任务具有最小裕度相近或相等的情况下，LSF 调度算法会产生严重的颠簸现象，引起大量的、不必要的任务切换。因为调度程序会消耗 CPU 带宽，保存任务上下文等信息会增加内存开销，这些使得系统性能大幅度下降，因此限制了 LSF 算法的应用。

2.2　LSF 算法的改进

为减少颠簸现象造成的系统资源的浪费，需要对 LSF 算法进行改进，尽量减少任务间的相互抢占，使之一个接一个地连续执行。但不能简单地通过临时设定不可抢占来实现，因为在任务的裕度降为零时，采用非抢占方法可能会使任务错失截止期[5]。因此，提出了一种新的调度模型：每个任务除了按裕度分配有优先级外，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 资源，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他任务抢占后，再恢复到原来的优先级。设定任务的优先级值越大，其优先级就越高，任务的抢占阈值(h)大于其优先级值(p)。假设任务 T_i 的优先级值和抢占阈值分别是 p_i 和 h_i，任务 T_j 的优先级值和抢占阈值分别是 p_j 和 h_j，当 T_i 占用 CPU 资源时，T_j 等待执行，这时 T_i 的优先级可看成是提高到其抢占阈值的等级(其实际优先级值仍然是 p_i)。随着时间的推移，T_j 的裕度逐渐减小，从而 p_j 逐渐增大，当 $p_j>p_i$ 时，T_j 的优先级值已经超过 T_i 的优先级值，这时，若按 LSF 调度策略，T_j 将抢占 T_i 执行，但采用抢占阈值策略后，T_i 的优先级已经提高到 h_i 层次，只有当 $p_j>h_i$ 时，T_j 才可抢占 T_i，从而延缓了 T_j 抢占 T_i 的速度。前述的 3 个任务采用改进策略之后减少了颠簸现象，如图 2 所示。

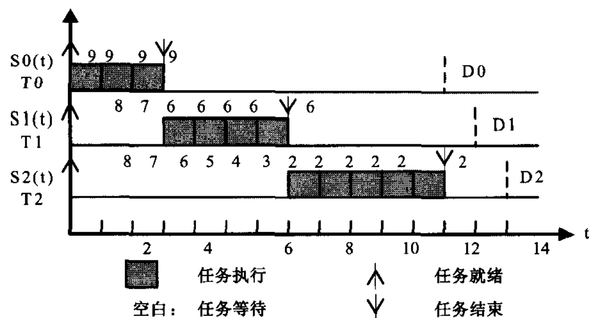

图 2　基于抢占阈值的 LSF 算法

这种基于抢占阈值的调度策略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。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。

2.3　抢占阈值的设定

抢占阈值的设定是改进算法的关键，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。根据前面关于优先级值越大，优先级就越高的假设，任务的抢占阈值需要设计成不小于相应的优先级，即 $h\geqslant p$。如果取 $h\geqslant p_{max}$，则调度策略演变为非抢占模型，因此在改进算法中，抢占阈值的取值范围为(p,p_{max})。基于抢占阈值的调度算法有很多，属于数学的分支——排队论的研究范畴，不仅应用于操作系统领域，在计算机网络、管理科学、控制理论等领域也都有诸多的应用，因此是当今理论界研究的热点。当前最新的研究大多基于模糊集理论，但由于调度程序在实时操作系统中运行极为频繁，其本身在系统中的开销不可忽视，过于复杂的阈值算法反而会影响到操作系统的总体性能，这也是许多优秀的算法无法被实时操作系统所采用的原因。为达到总体性能的优化，须避免过于复杂的阈值计算。本文采用了一种简化的处理函数：$h=ceil(K\cdot p)$，详见第 3 节。

2.4　可调度性分析

在改进算法中，设 u 为任务 T_i 的一个实例，S_i 为任务 T_i 的最坏情况开始时间，J_i 为任务 T_i 颠簸切换的时间，F_i 为任务 T_i 的最坏情况完成时间，一个任务最坏情况阻塞时间为

$$B_i=\max_{\forall j:P_j\geqslant P_i} X_j \tag{2}$$

则有

$$S_i(u)=B_i(u)+u\cdot X_i+\sum_{\forall j:P_j>P_i}\left(1+\left\lfloor\frac{S_i(u)+J_j}{T_j}\right\rfloor\right)\cdot X_j \tag{3}$$

$$F_i(u)=S_i(u)+X_i+\sum_{\forall j:P_j>P_i}\left\lceil\frac{F_i(u)+J_j}{T_j}\right\rceil-\left(1+\left\lfloor\frac{S_i(u)+J_j}{T_j}\right\rfloor\right)\cdot X_j \tag{4}$$

任务 T_i 最坏情况响应时间 R_i 为：

$$R_i=\max_{\forall u:0\leqslant u\leqslant Q}(F_i(u)+J_i-u\cdot t_i) \tag{5}$$

基于抢占阈值的调度模型是抢占和非抢占调度模型的泛化，它的可调度性不会差于抢占或者非抢占调度。但是，由于动态优先级的特点以及调度中依然无法完全消除颠簸现象，可调度性的判定存在潜在的不精确因素，系统任务的可调度性需由最坏响应时间 R_i 及具体应用要求决定。

3　算法在 Linux 中的实现

Linux2.6 调度程序由于引入了 O(1)调度算法以及全新的运行队列(runqueue)结构，使 schedule()变得简单，减少了锁操作，任务的优先级也不再集中计算，调度效率大大提高。基本流程可概括为切换前处理、选取候选任务、任务切换和切换后处理四个步骤。为保持现有 Linux 调度系统的架构，不破坏其 O(1)算法、优先级队列、优先级表示方法等特性，需要巧妙引入裕度，主要改进如下。

(1) 在任务控制块 task_struct 中增加"裕度"相关的属

性：任务剩余执行时间 *Xtime*(初值为估计执行时间)、任务提交时间 *Rtime*、相对截止期 *Dtime*、裕度值 *Stime*。设 T 为当前时间，由式(1)可得

$$Stime=(Rtime+Dtime)-T-Xtime \quad (6)$$

(2) 保持 Linux2.6 系统的运行队列(runqueue)结构以及候选(next)任务的选取方法，实时任务的优先级(prio)属性值一经初始设定后不再改变，当前(prev)实时任务的优先级按抢占阈值计算。在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，而非原来的 FIFO 形式。这样，新的实时任务调度的依据变为：任务的优先级越大，越先得到调度。当前实时任务的优先级按抢占阈值计算(由 *ceil* 函数处理)，优先级相同的任务，裕度值越小，越先得到调度。原有系统中，enqueue_task()函数实现了将进程插入到相应优先级队列的末尾，须对其进行改进，使其按裕度值大小顺序进行插入操作。

(3) 实时任务调度的动态性体现在裕度值的不断变化上，因此需要在时钟中断子程序 scheduler_tick()中增加对裕度值的实时更新处理。由式(6)可知，当前进程的剩余执行时间 Rtime 随时钟嘀哒而相应减少，其裕度值 Stime 不变；其他就绪进程剩余执行时间 Rtime 不变，但裕度值 Stime 却随时钟嘀哒而相应减少。此外，基于抢占阈值的候选任务的选取操作也要在 scheduler_tick()中做相应的修改。

(4) 对于按时间轮转方式运行的 SCHED_RR 实时任务，由于优先级队列的有序性，使得用完时间片的 SCHED_RR 任务不能再以插到队列尾的方式保留在 active 数组中，而需要插入到 expired 数组中。

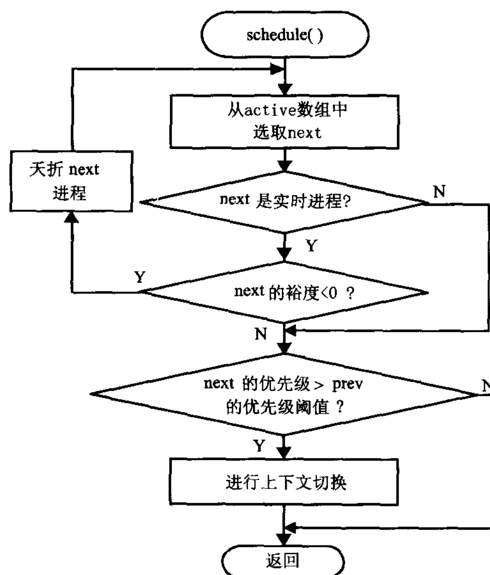

图 3　改进后的调度函数流程图

(5) 整个改进过程集中在候选任务选取阶段，系统不再是简单地选取第一个就绪任务作为候选任务，而要在此基础上进行必要的筛选：对于实时的候选任务，由于引入了裕度和抢占阈值，只有其裕度属性值大于 0，方可作为合格的 next(否则夭折该任务而重新一轮选择)，然后将其优先级与当前任务的抢占阈值比较，抢占阈值 $h=ceil(K \cdot p)$，其中比例系数 $K\in[1,2)$ 为给定常数，p 为与任务的裕度相对应的优先级，函数 $ceil(x)$ 表示取大于 x 的最小整数。如果候选任务的优先级大于当前任务的抢占阈值，则与当前任务切换；对于非实时的候选任务，由于可能存在处于 expired 数组中的 SCHED_RR 实时任务，为了保证实时任务的优先调度，只有在无实时任务的情况下方可作为合格的 next，否则需要先调换 expired 数组中实时任务部分而重新一轮选择。

改进后的调度函数流程如图 3 所示。

4　性能分析

4.1　保持 O(1)调度特性

实时任务“裕度”属性的引入，是在保持 Linux2.6 调度系统的可运行队列 runqueue、优先级数组的结构下进行的：当前实时任务的优先级按抢占阈值计算，原来实时任务的优先级 prio 变成了任务的静态优先级，依然保持不变，取值范围仍旧保持在[0,99]，它用来决定实时任务插入的优先级队列；裕度值 *Stime* 变成了任务的动态优先级，随时钟嘀嗒而发生改变，它用来决定任务插入队列中的位置。因此，对于任务优先级的计算、候选任务的选取和时间片的计算几个方面依然保持了分散计算的 O(1)调度特性。

4.2　实时调度的灵活性

正如前面分析，在本文采用的方法中，实时任务调度的动态性体现在裕度值的变化上。而基于抢占阈值的调度策略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，又减少了颠簸现象的产生，大大增强了实时任务的可调度性，而且结合 Linux2.6 的非实时任务调度，用户可以根据需要，灵活地选择是否采用新调度算法。

4.3　新调度的代价

等待中的实时任务裕度值动态变化的特点，造成在每个时钟中断中都必须对等待的实时任务的裕度值进行更新以及抢占阈值的计算，这个过程要遍历整个实时就绪队列。因此，当实时任务很多的情况下，这些计算将增加时钟中断的负担，降低系统的性能。但是，相对于实时任务调度性能的提升，这样的代价还是可以接受的。

结束语　针对 LSF(最小裕度优先)算法在实际应用中的缺陷以及 Linux 调度程序的特点，通过采用基于抢占阈值的 LSF 算法，并与 Linux 的运行队列(runqueue)、优先级数组(prio_array)等数据结构的巧妙结合，使得原本静态的实时任务调度变成动态调度。分析表明，新的调度系统能够很好地保证实时任务对调度的精确要求，有效地提高了任务的实时调度性能，同时大大减小了颠簸现象造成的资源消耗。在保持了原有系统 O(1)调度特性的同时，进一步提高了系统的实时性。

参考文献

[1] Liu C L, Layland J W. Scheduling algorithms for multiprogramming in a hard real-time environment[J]. The Association for Computing Machinery, 1973, 20(1): 46-61

[2] Hildebrandt J, Golatowski F, Timmermann D. Scheduling Coprocessor for Enhanced Least-Laxity-First Scheduling in Hard Real-Time Systems // Proc. of the 1lth Euromicro Conf. 011 Real-Time Systems. Los Alamitos: IEEE Computer Society Press, 2002: 208-215

[3] Terrasa A, Garcia - Fornes A, Botti V J. Flexible Real - Time Linux: A Flexible Hard Real-Time Environment [J]. Real-Time Systems, 2004, 22(2): 151-173

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

[5] 许占文，李歆. Linux2.6 内核的实时调度的研究与改进[J]. 沈阳工业大学学报，2006(8): 438-441