文章编号:1671-4598(2020)11-0238-04 DOI:10.16526/j.cnki.11—4762/tp.2020.11.048 中图分类号:TP39 文献标识码:A

基于 Linux 的多核实时任务调度算法改进

陈国良，朱艳军
（武汉理工大学 机电工程学院，武汉 430070）

摘要：嵌入式实时系统通常被实现为多任务系统，以满足多个外部输入的响应时间的最后期限约束；Linux 内核中已经实现了基于 EDF（earliest deadline first）调度算法的 DL 调度器，使得实时任务能在截止期限内运行完成；但对于多核处理器，由于实时任务在 EDF 算法下会出现 Dhall 效应，文章对 Linux 内核中实时任务调度算法进行了改进；在 EDF 算法的基础上，实现 LLF（least laxity first）调度算法并对其加以改进，通过降低任务上下文切换频率以及减少松弛度的计算来减小调度过程中的颠簸现象；实验证明该方法既避免了 Dhall 效应，又减少了任务上下文切换带来的系统开销，并使得任务能在截止期限内完成调度，取得了较好的调度性能。

关键词：实时任务；Linux 内核；多核处理器；LLF

Improved Multi—core Real—time Task Scheduling Algorithm Based on Linux

Cheng Guoliang，Zhu Yanjun
（School of Mechanical and Electronic Engineering，Wuhan University of Technology，Wuhan 430070，China）

Abstract：Embedded real—time systems are often implemented as multi—tasking systems to meet deadline constraints on the response time of multiple external inputs. The DL scheduler based on the EDF（Earliest Deadline First）scheduling algorithm has been implemented in the Linux kernel，so that real—time tasks can be completed within the deadline. But for multi—core processors，real—time tasks will have Dhall effect under the EDF algorithm. In view of the above problems，this paper proposes an improved method for real—time task scheduling algorithms in the Linux kernel. Based on the EDF algorithm，the LLF（Least Laxity First）scheduling algorithm is implemented and improved. It reduces the jitter in the scheduling process by reducing the task context switching frequency and reducing the slack calculation. Experiments show that this method not only avoids the Dhall effect，but also reduces the system overhead caused by task context switching，and enables tasks to be scheduled within deadlines，and achieves better scheduling performance.

Keywords：task scheduling；Linux kernel；multi—core processor；LLF

0 引言

随着计算机的发展，实时调度被广泛应用于实时性强的众多领域，如电子商务、交通管制、航空航天等，要求操作系统能够及时高效地处理多任务的运行。在该领域中，抢占式调度方式被普遍使用，然而抢占多任务时往往会增加内存的总开销和浪费 CPU 的带宽。

实时调度算法主要有 RM（Rate—Monotonic）调度[1]、EDF 调度[2]、LLF 调度[3]等，其中 EDF 调度算法从 Linux—3.14 版本开始加入调度策略。RM 与 EDF 算法在单核方面调度性能最佳，但在多核调度中存在 Dhall 效应，即存在正规化资源利用率总和任意接近于 0 的任务集不可调度。Linux 中可以通过区域划分的方式避免 Dhall 效应，但需要人为地划分任务允许运行的 CPU 核。LLF 算法可以规避 Dhall 效应，其根据任务紧迫程度来决定实时任务的调度顺序，任务松弛度越小表示任务越紧急，但当同一就绪队列中两个任务的松弛度比较接近时，可能会发生任务之间的频繁抢占。文献［4］提出使用 SchedISA 调度集，将处理器核心分组实现实时任务与非实时任务在不同的核心执行，来提高 EDF 调度实时性。文献［5—6］提出在就绪队列中在没有松弛度为零的任务时，不抢占当前任务，以此来降低任务上下文切换频率，减小系统开销。文献［7］提出了 LLF 与 LMCF（最低内存消耗优先）相结合的实时调度算法，当所有任务都相对截止期限松弛度不为 0 时，从 LLF 调度切换到 LMCF 调度。文献［8］提出在 SCHED_RR 调度策略与完全公平调度策略结合，依据任务权重参数来分配时间片大小，不能保证任务在期限内完成。其中，文献［4］和［9］均通过内核分组的方式提高 EDF 算法的调度性能，但在多核处理器中 EDF 算法相比 LLF 算法较差。文献［5-7］均是在 LLF 算法的基础上作出了相应改进，但在任务集可调度的前提下，同一就绪队列中出现多个等待任务的松弛度同时减为零时，任务可能不能在期限内调度执行完。

针对以上问题，下文基于现有 Linux 内核中的 EDF 调度算法实现原理，对传统 LLF 调度算法提出改进，主要通过减小上下文切换次数以及松弛度的计算，在避免 Dhall 效应的情况下最大程度减小了系统开销。下文描述基于以下假设：

收稿日期:2020-04-17； **修回日期**:2020-05-15。
基金项目:国家自然科学基金(61672396)。
作者简介:陈国良(1972-)，男，湖南湘潭人，副教授，主要从事智能控制与机器人技术、计算机视觉、机电一体化方向的研究。

1）系统中只有一个处理器；

2）在执行状态下，系统中每个任务的所有部分都可能被剥夺执行处理器的权力；

3）所有任务之间相互独立，并且无序；

4）任何任务都不能自己挂起。

1　Linux 实时调度问题

实时任务是指任务执行能在截止期限内完成，按照任务周期可以分为周期性任务、偶发性任务和非周期性任务[10]。对于偶发性与非周期性任务，其周期为相邻任务之间到达时刻的最小时间间隔，所以一般使用任务运行时间、截止期限、周期来描述一个实时任务。在 Linux 内核中实时任务主要通过 DL 调度类创建，与其他调度类相比具有更高的优先级，是基于 EDF 调度算法实现。实时任务集只有在可调度的情况下才能在满足期限约束，因此有如下定义。

定义 1：假设 M 个周期性实时任务运行在 N 个 CPU 核的处理器上，每个任务的处理时间是 C_i，周期时间是 P_i，$1 \leqslant i \leqslant M$。若任务利用率总和不超过 N，则任务集可调度，即：

$$\sum_{i=1}^{M} \frac{C_i}{P_i} \leqslant N \tag{1}$$

EDF 是一种基于截至时间最短优先调度的算法，每一个实时任务包含 3 个参数：$WCET$、D、P，其中 $WCET$（worst－case execution time）是任务在最坏情况一个周期运行的时间，D 是相对截至期限，P 是周期时间，因此实时任务可以表示为（$WCET$，D，P）。在 Linux 内核中，EDF 属于全局调度，任务可以在任意 CPU 核上执行，必要时进行核间迁移，提高了多核 CPU 的整体利用率。绝对截止期限等于相对截至期限加上 CPU 墙上时间，任务按绝对截止期限的大小排列，系统总是选取截止期限最小的任务运行。

例子 E_1：假设在 M 个 CPU 上，有 $M+1$ 个实时任务需要运行，任务描述如下：

第一个任务 $T_1 = (P, P, P)$；

剩余 M 个任务 $T_i = (e, P-1, P-1)$.

其中 e 是具有任意小的最坏情况任务运行时间，一次调度就能运行完。在某一时刻 t_0 这 $M+1$ 个任务被同时激活，因为第一个任务的截止期限等于 $t_0 + P$，而后面 M 个任务的截止期限都是 $t_0 + P - 1$，按照 EDF 调度规则，后面 M 个任务会优先在 M 个 CPU 上调度，而第一个任务需要等待时间 e 后才能被调度，执行完任务 1 后的时刻位于（$t_0 + e + P$），超过了其截止期限 P，如图 1 所示。但如果能在最开始分配一个 CPU 给任务 1 运行，那么 $M+1$ 个任务都能在截止期限内运行。

2　实时调度策略

在多核处理器系统中，LLF 调度比 EDF 具有更好的调度性能[7]。LLF 算法一定程度上可以解决多核中出现的 Dhall 效应，任务加入就绪队列时松弛度 $S = D - WCET$，如果得不到及时调度或任务已经运行了一部分时间，松弛度 S ＝ 绝对截止时间 － 当前时间 － 任务剩余运行时间。其中，松弛度最小的任务获得优先调度，当松弛度为 0 时任务立即被调度；当出现两个任务的松弛度相同时，按“最近最久未调度”原则调度。那么，例子 E_1 中的（$M+1$）个任务的松弛度分别是：

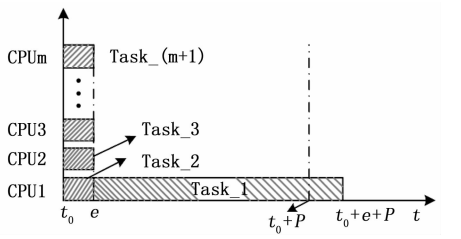

图 1　Dhall 效应

$$S_1 = 0, S_i (i = 2, 3, \ldots, m+1) = P - 1 - e$$

按松弛度最小的调度顺序，任务 1 最先得到调度，在截止期限内（$M+1$）个任务都得到了调度。任务调度顺序如图 2 所示。

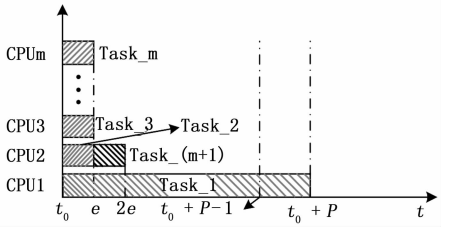

图 2　改进后的调度顺序

最小松弛度优先调度在多核情况下仍然不是最优的调度算法，当同一就绪队列上两个任务的松弛度比较接近时，会发生频繁的上下文切换；同时，每次 tick 周期到来都需要更新就绪队列中任务的松弛度，当系统中实时任务较多时，计算量相对较大。鉴于以上问题，提出以下改进方法：

1）只有当任务被激活加入就绪队列前或当前任务运行完触发主调度器时，更新该 CPU 就绪队列中所有任务的松弛度；

2）在每个 tick 周期仅更新待运行任务中松弛度最小任务的松弛度；

3）没有任务被激活时，直到有任务的松弛度为 0 或当前任务运行完，否则不发生任务切换；

4）当有任务被激活或需要从就绪队列选取下一个待调度的实时任务时，考虑任务抢占或交换相邻松弛度任务的调度顺序。

以上方法 1）和 2）用于减小松弛度计算，方法 3）、4）用于减少任务上下文切换次数，其中方法 1）、2）、3）易于实现，为了讨论方法 4），在实时任务集可调度的条件下设计如下例子 E_2：在同一 CPU 就绪队列中有 3 个任务被同时激活，如下：

$T_1 = (0.05P, 0.5P, 0.5P)$

$T_2 = (0.05P, 0.5P, 0.5P)$

$T_3 = (0.6P, P, P)$

松弛度分别是：

$S_1 = S_2 = 0.45P$

$S_3 = 0.4P < S_1$

当它们同时被激活时，按 LLF 调度算法任务 3 优先得到调度。经过时间 S_1，任务 1 和 2 如果均未通过负载均衡迁移至其他 CPU 核，它们的松弛度将同时减为 0，任务 1 或 2 会抢占当前 CPU。由于它们松弛度都为 0，即最紧迫的任务，无论谁抢占当前 CPU 另一个任务都无法在截止期限内调度执行完。如果交换相邻松弛度任务 3 与 1（假设在就绪队列中，松弛度按 3－1－2 从小到大排列），按 1－3－2 顺序排列任务，则任务 3 运行 0.4 P 时间时任务 2 松弛度减小为 0 抢占任务 3，即使当前 CPU 核上任务不发生迁移，也能按 1－3－2－3 顺序在期限内保证 3 个任务调度执行完。

定义 2：如果任务剩余运行时间大于其松弛度，则该任务为大活任务，否则为小活任务。

定义 3：将任务抢占与交换相邻松弛度任务都称为任务交换，其中不发生任务交换运行的任务称为原始任务，任务交换后运行的任务称为插队任务。

以上例子 E_2 中，按照定义 2 在任务刚被激活时，任务 1 和 2 都是小活任务，任务 3 是大活任务。从以上分析看出，先调度松弛度较大的小活任务可能减少任务上下文切换次数。由此提出以下任务交换策略：在任务交换时机到来时，有原始任务 $K(R_k,D_k,P_k,S_k)$，插队任务 $Q(R_q,D_q,P_q,S_q)$，如果 K 是大活任务且 Q 是小活任务，并有 $R_k > S_q$ 与 $R_q < S_k$，则优先调度任务 Q。

接下来对以上调整策略进行分析，如图 3 所示，以任务 K 为小活任务和任务为 K 大活任务两种情况讨论。若任务 K 为小活任务，有 $R_k < S_k < S_q$，在任务 K 运行完后，无论任务 Q 是大活任务还是小活任务其松弛度均大于 0，不必交换调度顺序。若任务 K 为大活任务，且任务 Q 为小活任务，需要加上条件 $R_k > S_q$ 与 $R_q < S_k$，才能满足交换条件。若任务 K 为大活任务，且任务 Q 也为大活任务，有 $R_q > S_q > S_k$，如果交换任务 K 与 Q 调度顺序，在不考虑任务在核间迁移的情况下会使任务 K 在 Q 运行完之前松弛度减为 0。

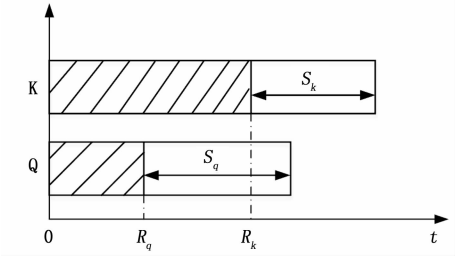

图 3 任务 K 与 Q 时间参数

3 实时调度实现细节

本节将说明如何在实时调度实体中调度 ILLF（改进的 LLF）任务，对 Linux 内核进行必要修改并包括其他系统调用，可以从 SCHED _ DEADLINE 调度类中获取新实时调度类的实现，并且新调度类在内核所有调度类中具有最高优先级。

在任务的创建阶段，新创建任务的 $WCET$，截止期限和周期作为参数传递，新任务通过这些参数计算其初始松弛度。更新新加入任务所在就绪队列上所有调度实体的松弛度，按松弛度大小将新任务实体 new _ task 加入就绪队列。其中，入队列函数除了使用 rb _ leftmost 与 rb _ left2most 分别记录松弛度最小的任务 M 与松弛度仅大于 M 的任务。Algorithm 1 说明了入队列过程，新任务实体所在就绪队列的根节点是 rb _ root _ of _ new，当新任务实体松弛度最小或仅大于当前 rb _ leftmost 对应调度实体时，需要更新记录值，最后函数 enqueue _ sched _ entity 将 new _ task 加入就绪队列。

```
Algorithm 1:
function
enqueue_laxity_entity_function(new_task) do
leftmost ← 1;
link ← rb_root_of_new;
while link do
parent ← link;
if new_task. laxity < parent. laxity then
link ← parent. letf;
else
link ← parent. right;
leftmost ← 0;
end if
end while
if leftmost=1 then
rb_left2most ← rb_leftmost;rb_leftmost ← new_task;elseif par-
ent = rb_leftmost then
rb_left2most ← new_task;
end if
enqueue_sched_entity (new_task, parent, link);
end function
```

在 CPU 当前任务执行完或有新任务加入就绪队列后，考虑任务交换。如下 Algorithm2 所示，参数 K、Q 分别是原始任务与插队任务。对于新加入任务，K 指当前任务，Q 指新加入任务；对于当前任务执行完，K 指 rb _ leftmost 所指任务，Q 指 rb _ left2most 所指任务。其中 big _ load=1 表示任务是大活任务，下一时刻调度函数 pick _ next _ laxity _ entity 返回的调度实体。

```
Algorithm 2:
function pick_next_laxity_entity(K,Q) do
if K. big_load = 1 and Q. big_load = 0 then
if K. runtime > Q. laxity and K. laxity ⩾ Q. runtime then
return Q;
end if
end if
return K;
end function
```

每个 tick 周期到来时，仅更新等待运行任务中松弛度最小的任务的松弛度。rb _ leftmost 指针关联的任务如果是当前任务，则待运行任务中松弛度最小的任务与 rb _

left2most 指针相关联，伪代码如下：

```
left_task ← rb_leftmost;
if left_task = cur_task then
left_task ← rb_left2most;
end if
left_task. laxity ← left_task. deadline − left_task. runtime − cur_time;
```

如下 Algorithm 3，当任务执行完，函数 dequeue _ sched _ entity 从就绪队列移除当前任务 cur _ task 时，需要更新 rb _ leftmost 与 rb _ left2most 记录的任务。

```
Algorithm 3:
function dequeue_laxity_entity(cur_task) do
if rb_leftmost = cur_task then
rb_leftmost ← rb_left2most;
rb_left2most ← rb_left2most. next;
else if rb_left2most = cur_task then
rb_left2most ← rb_left2most. next;
end if
dequeue_sched_entity (cur_task);
end function
```

4　实验结果与分析

为了验证上述 ILLF 调度算法效果，选择标准 Linux－3.14 版本内核，在拥有四核同构处理器的 tiny4412 开发板上进行实验。其中，标准 Linux 为了避免实时任务长时间占用 CPU，默认情况下在 1 s 时间内留出 5％的 CPU 时间给非实时任务。实验选择一组运行时间在［5 ms，80 ms］上的周期任务，最大任务利用率不能超过 4 * 95％＝380％。考虑到 Linux 内核中中断、软中断、自旋锁以及任务上下文切换带来的系统开销，任务利用率控制在 4 * 80％＝320％以内。

为了简化，假设每个任务的截止期限与周期相等，任务运行时间为毫秒的整数倍，并配置 Hz 等于 1 000（tick 周期时间是 1 ms）。创建 16 个实时任务：

$T_i = ((5i)ms, D_i, P_i)(i = 1,2,\ldots,16)$

分别考虑 2 种情况下任务调度：1）每个任务的任务利用率相同，松弛度差别较大；2）每个任务的松弛度相同，任务利用率不同。

定义 4：在 1 s 时间内平均每个核上任务上下文切换次数称为任务上下文切换频率。

使用 vmstat 命令监控任务活动，其中 cs 列显示每秒进程上下文切换次数。对于情况 1），每个 CPU 平均任务利用率分别从 10％～80％，EDF、LLF 与 ILLF 算法的任务上下文切换频率如图 4 所示，由图可以看出，在处理器任务利用率较低的情况下，3 种调度算法对应的上下文切换次数基本一样，在任务利用率达到 60％以上时，改进的 LLF 调度算法的优势得到显现。

对于情况 2），松弛度为 S_0，16 个实时任务分别为：

$T_i = ((5i)\text{ms}, (5i)\text{ms} + S_0, (5i)\text{ms} + S_0)$ 总任务利用率：

$$U_t = \sum_{i\leqslant 16}^{i=1} \frac{5i}{5i + S_0}$$

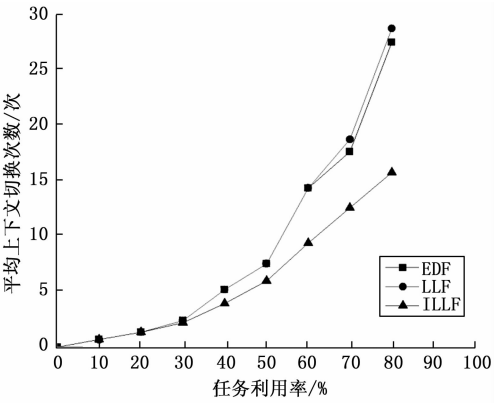

图 4　不同任务利用率上下文切换频率

总任务利用率最大 320％时，对应最小松弛度约为 180 ms，取松弛度为 200～270 ms，EDF、LLF 与 ILLF 算法任务上下文切换频率如图 5 所示，由图可以看出，随着松弛度的变化，平均上下文切换次数变化不大，但由于实时任务集的松弛度相同，LLF 调度算法带来了频繁的上下文切换，远多于 EDF 调度算法和 ILLF 调度算法带来的上下文切换次数。

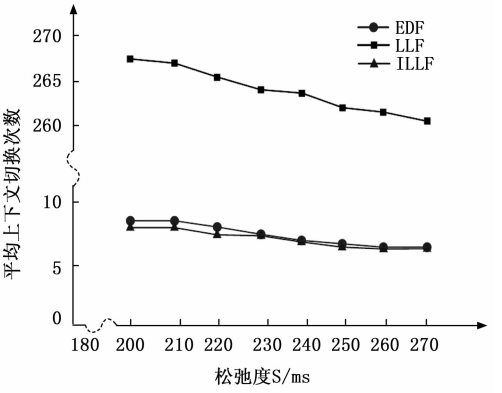

图 5　不同松弛度上下文切换频率

上面实验例子中的任务都是小活任务，为了体现改进方法 4）带来的影响，依此创建并激活如下任务：

$T_i(i = 1,2,3,4)$ ＝（60 ms，100 ms，100 ms）；

T_j（j＝5，6，...，12）＝（5 ms，60 ms，60 ms）。

在 EDF、LLF 与 ILLF 算法调度下任务上下文切换频率如表 1 所示，在 Linux 负载均衡的作用下，任务 T_1，T_5，T_9 被放在同一 CPU 核上运行，在仅使用方法 3）时，在任务 T_1 运行过程中，任务 T_5 和 T_9 的松弛度同时到达 0，没办法在截止期限内完成任务，而方法 4）使得任务 T_5 和 T_9 在任务 T_1 之前运行，既保证了任务能在截止期限内完成，又减少了任务上下文切换次数。

表 1　上下文切换次数频率

算法	EDF	LLF	ILLF
频率	34	207	23

（下转第 246 页）