基于EDF算法的可行性判定及实现

洪艳伟[1,2],赖 娟[2],杨 斌[1]
(1.西南交通大学 信息科学与技术学院,四川 成都 610031;
2.乐山师范学院 计算机科学系,四川 乐山 614004)

摘 要:实时调度算法是实时系统中的关键技术。验证实时调度算法的可行性是在实时系统中实施某种调度算法的必经环节。在介绍实时系统中常用的各种实时调度算法,包括固定优先级调度算法和动态优先级调度算法基础上,详细分析了动态优先级调度算法EDF算法的运算过程和使用条件。提出了该算法在实际应用中存在的问题。针对该硬实时调度算法,提出了分别在简单模型上和复杂模型上如何判定实时任务的可行性。为实际应用中实现该实时调度算法确定了依据。

关键词:实时;调度算法;可行性分析;响应时间
中图分类号:TP316 **文献标识码**:A **文章编号**:1673-629X(2006)11-0097-03

Feasibility Test and Realization Based on EDF Algorithm

HONG Yan-wei[1,2], LAI Juan[2], YANG Bin[1]
(1.School of Information Science &Technology, Southwest Jiaotong University, Chengdu 610031, China;
2.Department of Computer Science, Leshan Teachers College, Leshan 614004, China)

Abstract: The real time scheduling algorithm plays an important role in the real time systems. The test for scheduling algorithm is a very important step . The static priority scheduling algorithm and the dynamic priority scheduling algorithm are two kinds of scheduling algorithms on real time systems. Analyses the condition and the problem for EDF scheduling algorithms. Present a method for testing scheduling algorithm based on the response time of tasks in the worst condition for simple model and complex model. Determine the basis to realize this real-time scheduling algorithm for the practical application.

Key words: real-time; scheduling algorithm; feasibility analysis; response time

0 前 言

检查一组或一个具体实时任务是否能采用某种调度算法进行实时调度,这一过程叫调度算法的可行性判定。硬实时调度算法,分为速率单调调度法(RM)、时限调度法(DM)和最早时限优先调度算法(EDF)。EDF调度总是比固定优先级调度取得更高的利用率。相对于RM和DM,EDF的时限是任意的,更适用于实际的实时系统[1]。

针对不同的算法,可行性测定有很多种[2]。文中对常见的硬实时调度算法EDF介绍了相应的可行性测定方法,即假定任务同时启动,任务的最坏响应时间(即任务就绪到任务被完成的最长时间段)是否大于其任务时限,若大于,则调度可行,否则任务不可调度。

1 EDF简单实时模型的可行性测定

简单实时模型即各任务都是独立的周期任务,任务之间相互独立,无资源共享。Liu和Layland的论文[3]不仅介绍了针对固定优先级调度的基于利用率的测试,而且也为最早时限优先级算法(EDF)给出了一个基于利用率的测试:只要进程集合(对于简单进程模型)的利用率 $\sum \frac{C_i}{T_i}$ 小于1,那么所有时限将能被满足。但这种测试方法的缺点是不准确,不适应于更一般的进程模型。

2 EDF的基于最坏响应时间的可调度性测试

所谓响应时间是指任务的某次释放到任务完成所花费的时间,而最坏响应时间[4]是指所有响应时间中最长的响应时间。文中介绍的方法——EDF的基于最坏响应时间的可调度性测试,即是首先找出所给任务集的各个任务的最坏响应时间,然后将各个任务的最坏响应时间与相对时限相比较,如果所有任务的最坏响应时间都不大于各自的相对时限,那么该任务集可调度。如果没有通过测试,则该任务集不可调度。对调度的判断采用响应时间测试方法的优点之一是该方法适应于更一般的进程模型;优点之二它们是充分必要条件——如果任务集合通过测试,则可调度,否则,不可调度。可见找出最坏响应时间对于任务的可调度性的判断是非常重要的。下面介绍如何找出任务的最坏响应时间。

收稿日期:2006-01-20
作者简介:洪艳伟(1974-),男,四川眉山人,硕士研究生,讲师,主要研究方向为实时系统、嵌入式系统;杨 斌,副教授,研究方向为操作系统、实时系统、嵌入式系统。

EDF 调度算法的最坏时间响应分析的基本思想是：

所求任务 i 的最坏响应时间一定在处理器的忙碌期找到[4,5]。可知任务 i 的最坏响应时间不一定在处理器的第一忙碌期找到，但它能在这样的忙碌期找到：在该忙碌期中，除了任务 i 外其它任务都在忙碌期的开始同时释放，然后各个任务按各自的周期而再次释放。因为虽然任务同时释放的情况很特殊，但可以把其它任务都移到忙碌期的开始时刻后再同时释放，这种移动不但不会减少任务 i 的响应时间，相反可能增加其响应时间，而我们就是要求任务 i 的最坏响应时间，因此这样的移动是合理的，更有利于计算出任务的最坏响应时间。

设忙碌期的始端时刻为 t_1，结束时刻为 t_2。除任务 i 外的任务于忙碌期的开始时刻同时释放，任务 i 的相对时限为 D_i，设任务 i 的某次执行于时刻 a 释放，因此任务 i 的绝对时限为 $d = a + D_i$。在忙碌期内所有任务的绝对时限都小于或等于任务 i 的绝对时限，因此任务 i 的完成时刻即是忙碌期的结束时刻 t。见图 1。

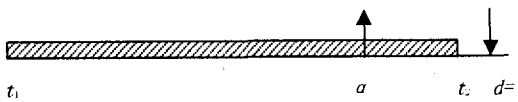

图 1 任务忙碌期示意图

为讨论方便，令 $t_1 = 0$，那么任务 i 的响应时间即是 $L_i(a) - a$。

现在的问题就变成如何计算忙碌期的长度 $L_i(a)$。

忙碌期长度即是从时刻 t_1 释放到时刻 t_2 完成的任务的计算时间总和。所以

$$L_i(a) = \sum_{D_j \le a+D_i}^{j \ne i} \min\left\{\left\lceil \frac{L_i(a)}{T_j} \right\rceil, 1 + \left\lfloor \frac{a + D_i - D_j}{T_j} \right\rfloor\right\} C_j + \delta_i(a, L_i(a)) C_i \quad (1)$$

[5]

其中，$\delta_i(a,t) =$

$$\begin{cases} \min\left\{\left\lceil \frac{L_i(a) - s_i(a)}{T_i} \right\rceil, 1 + \left\lfloor \frac{a}{T_j} \right\rfloor\right\}, & \text{if } L_i(a) > s_i(a) = 0 \\ 0, & \text{otherwise} \end{cases}$$

任务 i 的第一次释放时刻为 $s_i(a) = a - \left\lfloor \frac{a}{T_i} \right\rfloor T_i$。对于等式 $L_i(a) = \sum_{D_j \le a+D_i}^{j \ne i} \min\left\{\left\lceil \frac{L_i(a)}{T_j} \right\rceil, 1 + \left\lfloor \frac{a + D_i - D_j}{T_j} \right\rfloor\right\} C_j + \delta_i(a, L_i(a)) C_i$

可知，因为 $L_i(a)$ 是未知的（它是正在计算的值），由于高限函数和低限函数，使方程难以求解。可构造一递推关系

$$L_i^{m+1}(a) = \sum_{D_j \le a+D_i}^{j \ne i} \min\left\{\left\lceil \frac{L_i^m(a)}{T_j} \right\rceil, 1 + \left\lfloor \frac{a + D_i - D_j}{T_j} \right\rfloor\right\} C_j + \delta_i(a, L_i^m(a)) C_i$$

若令 $W_i(a,t) = \sum_{D_j \le a+D_i}^{j \ne i} \min\left\{\left\lceil \frac{t}{T_j} \right\rceil, 1 + \left\lfloor \frac{a + D_i - D_j}{T_j} \right\rfloor\right\} C_j + \delta_i(a,t) C_i$

那么等式(1) 变成

$$L_i^{(m+1)}(a) = W_i(a, L_i^{(m)}(a)) \quad (2)$$

(其中 $L_i^{(0)}(a) = \sum_{D_j \le a+D_i}^{j \ne i} C_j + I_{\{s_i(a)=0\}} C_i$)

经过证明只要利用率 $\sum \frac{C_i}{T_i} \le 1$，那么递推式(2) 就收敛，方程有解。

这样当 $L_i^{(m+1)}(a) = L_i^{(m)}(a)$，方程的解 $L_i(a)$ 就找到，就是 $L_i^{(m)}(a)$。

求解过程可以用时间窗口的概念来总结一下求 $L_i(a)$ 思路：可以将 $L_i(a)$ 这段时间称为时间窗口（见图 1）。那么在时间窗口内，只能有优先级高于或等于任务 i（即绝对时限小于或等于任务 i 的时限 d）的任务能执行，随着更多优先级的任务落入时间窗口，时间窗口的长度不断扩大，当没有更高优先级的任务需要执行时，则时间窗口到达最大，此时的时间窗口的长度即是忙碌期长度 $L_i(a)$。

知道了 $L_i(a)$ 的表达式后，可以在忙碌期内让 a 取不同的值，通过不同的 a 值，计算出任务 i 不同的 $L_i(a)$ 值，响应时间为 $r_i(a) = \max\{C_i, L_i(a) - a\}$，并从其中找出最大的 $r_i(a)$，这就是所要求的最坏响应时间 $r_i = \max\{r_i(a)\}$。

上述方法对于判定任务集的可行性相当有效。但由于计算响应时间的步骤多而且复杂，因此考虑用程序实现。程序 C 代码如下：

```
/* 首先定义一个实时任务结构体 rt_task_struct，其成员 T 为实时任务的周期，成员 D 为实时任务的绝对时限，成员 C 为实时任务的最坏执行时间。 */
typedef struct
    {
    int T;
    int D;
    int C;
    }rt_task_struct;
main(void)
{
    /* 定义实时任务结构体数组 rt_task[40]，定义最大任务数为 40 */
    rt_task_struct rt_task[20];
    int tasknum,;/* 本次待判定的任务数 tasknum */
    int r[40],ai[40];/* r[40]数组用来放置相应任务的最坏响应时间，ai[40]用来放置取得该响应时间时的时刻值 */
    float L0=0,Ln1=0,Ln=0;
    float sia,I_sia;
    float Lm0=0,Lm=0,Lm1,ria;
    float w,delta,wm=0;
    float m1,m2,m3,m4;
    int i,j;
    scanf("%d",&tasknum);/* 输入本次待判定的任务数 */
    for(i=0;i<tasknum;i++) /* 输入所有实时任务的周期、时限、最坏执行时间 */
```

```
        }
        scanf("%d,%d,%d",&rt_task[i].T,&rt_task[i].D,
&rt_task[i].C);
        }
    /* 计算最长忙周期 Ln1,因为实时任务的最坏响应时间一
定在最长忙周期中找到,利用公式 Ln1 = Σ(i=1..tasknum) ⌈Ln1/Tj⌉ * Tj */
    for(i=0; i<tasknum; i++)
        L0=L0+rt_task[i].C;
    Ln1=L0;
    while(Ln1! =Ln)
      {
      Ln=Ln1;
      Ln1=0;
      for(i=0;i<tasknum;i++)
        Ln1=Ln1+(float)ceil(Ln/rt_task[i].T)*rt_task[i].
C;
      }
    a_max=(long)Ln1;/* 将不大于最长忙周期 Ln1 的最大整
数赋给 a_max */
    for(i=0;i<tasknum;i++) /* 计算每个任务的最坏响应
时间 */
    {
    for(a=0;a<=a_max;a++)
      {
      sia=a-(float)floor((float)a/rt_task[i].T)*rt_task[i].
T;
      if(sia==0)
        1_sia=1;
      else
        1_sia=0;
      for(j=0;j<tasknum;j++)
      {
        if(j==i||(rt_task[j].D>a+rt_task[i].D)) contin-
ue;/* 如果任务 j=i 或者其时限 Dj>a+Di,则退出本次循环
*/
        Lm0=Lm0+rt_task[j].C;
        }
        Lm0=Lm0+1_sia*rt_task[i].C;
        Lm1=Lm0;
      while(Lm1! =Lm)
        {
        Lm=Lm1;
        Lm1=0;wm=0;
        if(Lm > sia)
         {
         m1=(float)ceil(Lm-sia/rt_task[i].T);
         m2=1+(float)floor((float)a/rt_task[i].T);
         delta=min(m1,m2);
         }
        else
        delta=0;
        for(j=0;j<tasknum;j++)
        {
        if(j==i||(rt_task[j].D > a+rt_task[i].D)) contin-
ue;
        m3=(float)ceil(Lm/rt_task[j].T);
        m4=1+(float)floor(((float)a+rt_task[i].D-rt_task
[j].D)/rt_task[j].T);
        wm=wm+min(m3,m4)*rt_task[j].C;
        }
        w=wm+delta*rt_task[i].C;
        Lm1=w;
        }
      ria=max(rt_task[i].C,Lm1-a);
      if(ria>r[i]) {r[i]=ria; ai[i]=a;}
      }
printf("%f,%f,%f,%f,%f,%f\n",i,rt_task[i].T,rt_task[i].
D,rt_task[i].C,r[i],ai[i]);
}}
```

下面举例说明:

例:假定实时系统中有如下 4 个任务,任务时间属性如表 1 所示(时间单位:ms)。

表 1　任务时间属性

i	D_i	T_i	C_i
1	4	4	1
2	9	6	2
3	6	8	2
4	12	16	2

由表 2 可知任务 3 的最坏响应时间为 $r_3 = 4$(表 2 中的数据为任务 3 对应于不同 a 值时的响应时间)。

表 2　任务 3 的各响应时间

$a = 0$	$r_3(a) = 3$
$a = 1$	$r_3(a) = 2$
$a = 2$	$r_3(a) = 2$
$a = 3$	$r_3(a) = 2$
…	$r_3(a) = 2$
$a = 8$	$r_3(a) = 2$
$a = 9$	$r_3(a) = 4$
$a = 10$	$r_3(a) = 4$
…	$r_3(a) = 2$

还可以按该方法计算出其它任务的最坏响应时间(见表 3)。

表 3　任务的最坏响应时间

任务 i	最坏响应时间
$a_1 = 11$	$r_1 = 2$
$a_2 = 6$	$r_2 = 7$
$a_3 = 9$	$r_3 = 4$
$a_4 = 10$	$r_4 = 3$

3　小　结

实时调度算法的可行性分析是实时系统的关键技术,

(下转第 102 页)

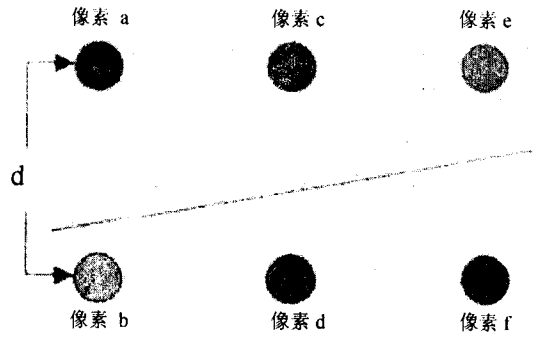

图 3 直线的反走样原理图

设相邻像素距离为 d，且图形共有 16 个灰度级别。像素 a 与理想线距离为 15/16 d，故 a 的亮度为 1，像素 b 的亮度则为 15。同理，像素 c 与理想线距离为 9/16 d，其亮度为 7，而像素 d 的亮度为 9。

可见，这种以理想线段为中心的像素对；按照亮度和等价于单个像素的方法绘制出来的线段，实际显示效果是一条在正确位置的光滑线段，从而实现了直线的反走样。

3 改进的 Bresenham 反走样算法

把上述两种算法结合起来，便可得到一种新的快速反走样算法。其最大的优点是在低分辨率的显示环境中获得非常好的效果以及更快的速度。欲绘制从 a 点到 b 点的线段，首先根据直线斜率 e，确定是以 x 轴还是 y 轴为基本轴进行画线。本图直线以 x 轴为对称轴，利用反走样的矩形滤波算法，x 轴在每次循环中，a 点和 b 点横坐标沿 x 轴一增一减，并且利用沿 y 轴的增量 e 来确定实际直线的两点，即一次主循环点亮 4 个像素点。直到把直线实际中心用两个像素点亮后，反走样直线绘制完毕。

如上所示，利用此方法，优化计算并进行反走样处理，加快了生成速度，尤其在绘制比较长的直线过程中，此算法的效率明显。

对于显示效果而言，由于反走样直线的灰度级越多，直线的平滑效果越好，但直线的整体亮度会减弱。实际在光栅扫描显示器上，像素的面积很小，并且人的眼睛对不同的灰度也很难分辨。因此没有必要设很多灰度级[4]。

如图 4 所示，设总共有 8 个灰度级别，即 $G = 8$。对于直线 a，在实际点 k_1，用像素点 A 和 B 来表示。在 k_1 处，沿 y 轴增量为 e_1，$1/8 < e_1 < 2/8$，判断 $e_1 < 3/16$，即 k_1 更靠近 1/8 处，于是 B 的灰度为 $7/8G$，A 的灰度级为 $1/8G$。同理，k_2 处可知 $5/8 < e < 6/8$，但更接近 6/8 处，于是 C 的灰度设为 $6/8G$，D 的灰度为 $2/8G$。

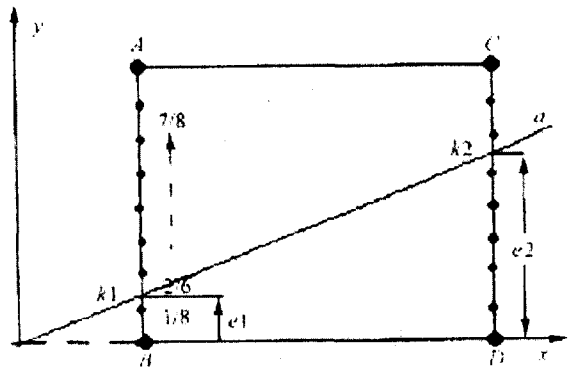

图 4 亮度计算方法

4 结 论

综上所述，通过对 Bresenham 算法的方法和其中参数的分析，改进生成一种新的反走样算法。本改进算法在配有 ARM7 微控制器 LPC2290 的 MagicARM2200 中得以实现[5]，实验证明可以明显加快反走样直线的生成速度，并且在低分辨率的显示环境中获得了非常好的效果[6]。

参考文献：

[1] Abrash M. 图形程序开发人员指南[M]. 北京：机械工业出版社，1998.

[2] 刘勇奎. 计算机图形学的基础算法[M]. 北京：科学出版社，2001.

[3] 钟奉金. VGA16 上个灰度级别的设置[J]. 电脑开发与应用，1995，8(1)：60－63.

[4] 孙德敏. 工程最优化方法及应用[M]. 合肥：中国科学技术大学出版社，2000.

[5] 刘勇奎，李 彬. 多灰度显示器上的曲线绘制[J]. 计算机工程与设计，1997，18 (4)：60－64.

[6] 倪明田，吴良芝. 计算机图形学[M]. 北京：北京大学出版社，1999.

(上接第 99 页)

决定了实时系统能否采用某种调度算法进行调度。文中采用了最坏响应时间的分析方法，即计算出系统中每个任务的最坏响应时间，然后判断每个任务的时限是否能满足条件，从而决定实时任务的可行性。当然，这种方法也在不断地发展和完善，人们为提高分析方法的灵活性，以及取消模型的更多限制而努力工作着。

参考文献：

[1] 翟鸿鸣. 单处理器系统的实时调度算法研究[J]. 微机发展，2003，13(10)：99－101.

[2] Joseph M，Pandya P. Finding Response Times in a Real－Time System[J]. The Computer Journal，1986，29(5)：390－395.

[3] Liu C，Layland J. Scheduling Algorithms for Multiprogramming in a Hard－Real－Time Environment[J]. Journal of the ACM，1973，20(1)：40－61.

[4] Sprunt B，Sha L，Lehoczky J. Aperiodic Task Scheduling for Hard－Real－Time Systems[J]. Real Time Systems，1989 (1)：27－60.

[5] Spuri M，Buttazzo G. Scheduling aperiodic tasks in dynamic priority systems[J]. Real－time Systems J，1996(1)：179－210.