文章编号: 1001-9081(2005)12Z-0447-02

一种改进的时间片轮转调度算法

肖建明, 张向利
(桂林电子工业学院 通信与信息工程系, 广西 桂林 541004)
(xjm_netease@163.com)

摘 要: 通过对时间片轮转调度算法中进程最后一次执行时间片分配的优化, 提出了一种改进的时间片轮转调度算法, 该算法具有更好的实时性, 同时减少了任务调度次数和进程切换次数, 降低了系统开销, 提升了CPU的运行效率, 使操作系统的性能得到了一定的提高。

关键词: 时间片轮转调度; 调度算法; 操作系统
中图分类号: TP316.2 文献标识码: A

时间片轮转调度算法作为一种经典的进程调度算法, 它最成功之处在于, 利用单CPU实现了分时多任务操作系统, 大大提高了CPU的吞吐量和利用率。而在实时操作系统中, 利用综合考虑优先级的时间片轮转调度算法, 可在保证实时响应性的前提下, 及时地对数据进行实时处理。本文则主要对时间片轮转算法作了进一步的研究, 提出了一种改进的时间片轮转调度算法。

1 算法原理

1.1 时间片轮转调度算法

时间片轮转法是特别适合于分时系统的一种调度算法。时间片轮转法[1]的基本思想是: 将CPU的处理时间划分成一个个时间片, 就绪队列中的诸进程按所分配的时间片轮流使用CPU资源。当分配的时间片用完时, 就被强迫让出CPU, 该进程进入就绪队列, 等待下一次调度。同时, 进程调度又去选择就绪队列中的一个进程, 分配给它一个时间片, 以投入运行。

采用此算法的系统的进程就绪队列往往按进程到达的时间来排序[2]。进程调度程序总是选择就绪队列中的第一个进程, 也就是说按照先来先服务原则调度, 但进程占有处理机后仅使用一个时间片。在使用完一个时间片后, 即使进程还没有完成其运行, 也必须释放出(被抢占)处理机给另一个就绪的进程, 该被抢占的进程则返回到就绪队列的末尾重新排队, 等候再次被调度运行。

1.2 改进后算法原理

改进后算法是在原算法原理基础上作了如下修改: 即进程在运行到即将执行完毕时, 通过增加一定的时间片值, 使得当前进程能不被调度程序切换出去而连续的执行完剩下的工作。其原理为: 在时间片轮转调度算法中对进程最后一次执行时间片进行优化, 即当进程按时间片轮转法调度时, 如果当前进程运行还需占用的CPU时间已不足进程分配到的时间片值的二分之一(此值可调整)时, 调度算法自动为当前进程增加一定的时间片值, 使之能继续获得CPU的使用权, 从而立即完成剩余代码的执行。

2 算法实现

下面以Linux0.11版相关部分源代码为例, 简单说明该算法的实现方法。

Linux操作系统是利用系统时钟中断来实现时间片轮转算法的。Linux中PC机的可编程定时芯片8253被设置成每隔10ms(我们常称之为1个系统滴答)就发出一个时钟中断(IRQ0)信号。因为每经过一个滴答就会调用一次时钟中断处理程序, 该处理程序会从被中断程序的段选择符中取出当前特权级(CPL)作为参数调用do_timer()函数。do_timer()函数根据特权级对当前进程运行时间作累计, 如果CPL=0, 则表示进程是运行在内核态时被中断, 同时把进程的内核运行时间统计值stime加1, 否则把进程用户态运行时间统计值utime加1。接着是用户定时器和软驱定时器处理程序, 最后会对当前进程运行时间进行处理: 把当前进程运行时间片(counter)减1, 若此时当前进程运行时间片还大于0, 表示分配的时间片还没有用完, 于是就退出do_timer()继续运行当前进程, 否则, 说明该进程已用完了分配的使用CPU的时间片, 程序再根据中断程序的级别来确定进一步的处理方法。若被中断的当前进程是工作在用户态, 则调用调度程序schedule()切换到其他进程去运行, 否则说明是处于内核态被中断, 则立即退出。当所有进程的时间片值均用完时, 在schedule()函数中会按优先级权值, 更新每一个任务的counter值后, 再按时间片轮转法调度。其部分源代码为[3]:

```
/*参数 cpl为当前特权级:0或 3, 0表示内核代码在执行
/*对于一个进程由于执行时间片用完时, 则进行任务切换, 并执行一个计时更新工作
Void do_timer(long cpl)
{
  extern int beepcount;
  extern void sysbeepstop(void);
  if(beepcount)
    if(!--beepcount)
      sysbeepstop();
  /*以上为控制扬声器发声部分代码
  /*如果当前特权级(cpl)为 0(最高, 表示是内核程序在工作), 则将内核态运行时间 stime递增, 如果 cpl>0, 则表示是一般用户程序在用户态下被中断, 增加 utime
  if(cpl)
    current->utime++;
  else
    current->stime++;
  /*处理用户定时器程序开始
  if(next_timer)
  {
    next_timer->jiffies--;
    while(next_timer&& next_timer->jiffies<=0)
    {
      void(*fn)(void);
      fn= next_timer->fn;
```

收稿日期: 2005-06-02 基金项目: 广西区教育厅基金资助项目(D200328)

作者简介: 肖建明(1973-), 男, 江西萍乡人, 硕士研究生, 主要研究方向: 基于Linux的嵌入式实时操作系统研究; 张向利(1968-), 女, 陕西渭南人, 副教授, 主要研究方向: 计算机软件及应用.

```
        next_timer->fn = NULL;
        next_timer = next_timer->next;
        (fn)();                          /*调用处理函数*/
    }
    sti();                               /*处理用户定时器程序结束*/
//下面两行为处理软驱定时程序
if(current_DOR & 0x0f)
  do_floppy_timer();
//当前进程运行时间片处理
if((--current->counter)>0)
  return;                /*如果进程运行时间还没完,则退出*/
current->counter=0;
if(!cpl)
  return;      /*在内核态下运行时不依赖counter值进行调度*/
schedule();
  //Linux中此调度器会选择时间片值最大者先被调度运行
}
```

基于上述算法之实现代码,改进后的调度算法实现方法有两种:

方法一:在 schedule()函数中,更新每一个任务的运行时间片(counter)值时,将当前进程的已运行时间与进程需运行的时间进行比较,若其差值比分配的运行时间片值的1.5倍小,则将 counter值增加当前值的一半。显然该方法的前提条件是:需要知道参与调度的进程运行所需的时间。

具体实现时只需在 schedule()函数里更新进程时间片值之 for循环的尾部增加如下语句(带下划线部分)即可:

```
//counter值的计算方式为 counter=counter/2+priority
for(p=&LAST_TASK; p>&FIRST_TASK; --p)
  if(*p)
    (*p)->counter=((*p)->counter>>1)+(*p)->
    priority;
  /*假定(*p)->etime为估计的进程运行所需时间*/
  if((*p)->etime-(*p)->utime-(*p)->stime-
  (*p)->counter<(*p)->counter>>1)
  (*p)->counter+=((*p)->counter>>1
}
```

方法二:此方法需配合应用编程来实现,即在内核中设置一全局变量(如 int ScheduleCounterNice=0),在应用编程时,当编写到应用即将完成之处的代码时,通过系统调用置全局变量 ScheduleCounterNice为适当值。在内核部分的其他修改则只需在上面的 do_timer()函数中带下划线行(current->counter=0;)前面增加下面语句则可:

```
if(ScheduleCounterNice)
{//为进程增加一定的运行时间片值
  current->counter+=ScheduleCounterNice
  return;
}
```

3 算法性能分析及意义

下面先从通用操作系统角度来对算法的性能进行分析:假设当前仅有三个任务 T1、T2、T3处于就绪态,其所需的处理时间分别为:101ms、202ms、303ms。操作系统进行进程切换所需花费的时间为 CST,调度延迟时间(即在某个调度时机,调度器选择下一个进程所用的时间)为 SDT,则三个任务在以下各条件下用改进前后之时间片轮转调度算法进行调度所得到的运行时间结果比较见表1。

表1 三个任务在算法改进前后的运行时间比较

		改进前运行时间	改进后运行时间	改进后提前完成时间
A	T1	301ms+30*(CST+SDT)	281ms+27*(CST+SDT)	20ms+3*(CST+SDT)
	T2	503ms+51*(CST+SDT)	483ms+46*(CST+SDT)	20ms+5*(CST+SDT)
	T3	606ms+52*(CST+SDT)+10*SDT	606ms+47*(CST+SDT)+9*SDT	5*(CST+SDT)+SDT
B	T1	606ms+32*(CST+SDT)	606ms+29*(CST+SDT)	3*(CST+SDT)
	T2	605ms+31*(CST+SDT)	595ms+28*(CST+SDT)	10ms+3*(CST+SDT)
	T3	603ms+30*(CST+SDT)	573ms+27*(CST+SDT)	30ms+2*(CST+SDT)
C	T1	191ms+9*(CST+SDT)	161ms+6*(CST+SDT)	30ms+3*(CST+SDT)
	T2	403ms+22*(CST+SDT)	383ms+17*(CST+SDT)	20ms+5*(CST+SDT)
	T3	606ms+23*(CST+SDT)+20*SDT	606ms+18*(CST+SDT)+19*SDT	5*(CST+SDT)+SDT

表1中:A为三个任务每次由调度程序所分配到的时间片均为 10ms条件下的结果;B为三个任务每次由调度程序所分配到的时间片分别为 10ms、20ms、30ms条件下的结果;C为三个任务每次由调度程序所分配到的时间片分别为 30ms、20ms、10ms条件下的结果。

从表1数据分析可知,当任务运行所需花费的时间与该任务被调度时被分配的时间片值之比的小数部份大于0且小于0.5(此值可按实际要求调整)时,用改进后的时间片轮转调度算法能够减少任务的实际完成时间,同时也减少了任务调度次数和进程切换次数,从而减小了系统开销,提升了 CPU的运行效率,使操作系统的性能获得了一定的提高。

另外,在实时操作系统中,对于同一优先级的任务用改进后的时间片轮转法进行调度,具有更为的重要意义,因为在实时系统中任务在规定的期限内能否执行完毕将可能导致灾难性的后果。从上面的例子分析中可知,用改进前的时间片轮转法算法对同一优先级任务进行调度时,当前任务可能会在仅差几个甚至一个时间片就能运行完毕时,却被进程调度程序切换出去,从而导致本可以在规定的期限内完成的任务最终没能在规定的期限内完成。而改进后的算法则能够避免这种可能性的发生。

4 结语

从上面的分析可知,改进后的算法是切实可行的,并且不管是在通用操作系统还是在实时操作系统中,都具有实际的应用价值,另外,该算法思想也能应用于其他一些调度算法的改进(如多级反馈队列调度算法[4]),从这个意义上说,该算法也具有一定的理论价值。

参考文献:

[1] SILBERSCHATZ A. Applied Opertion System Concepts[M]. John Wiley & Sons, Inc, 2001.

[2] 罗宇. 操作系统[M]. 北京:电子工业出版社, 2003.

[3] 赵炯. Linux内核完全注释[M]. 北京:机械工业出版社, 2004

[4] KLEINROCK L, MUNTZ RR. Processor Sharing Queueing Models of Mixed Scheduling Disciplines for Time Shared Systems[J]. Journal of the ACM, 1972, 19(3): 464-482.