**基于动态优先级的进程调度方案设计**

**摘要**

Linux具有内核源码公开，性能稳定等一系列优点，正迅速进入实时控制领域。但Linux并非真正的实时操作系统，所以必须对其进行实时提升。传统的动态优先级调度方案一般仅从进程的存储结构，动态优先级的计算方法入手。但若是时间片间隔设置不当的情况下，仍会造成一定程度的“饥饿现象”与额外的处理器开销。本文将时间片调整，动态优先级的计算方案设计及等待进程的存储结构三者结合起来，设计出改进方案，并在最后进行了性能分析。

**关键字：**时间片调整 动态优先级计算方案 存储结构 索引顺序表

1. **进程调度的基本原则**

在进行进程调度方案之前，首先要明确进程调度的基本原则。

进程调度即操作系统合理分配CPU时间以执行就绪进程。其关注如何使系统能够保持较短的响应时间和较高的吞吐量，以及如何在多个可运行的进程中选取一个最必须的进程投入运行。进程调度包括以下两个方面的内容——何时启动调度器和调度器应该执行怎样的调度算法。一般来说，一个好的调度算法应当考虑的主要因素包括：

公平性：保证每个进程都得到合理的CPU时间

高效性：使CPU保持忙碌状态，总是有进程在CPU上运行

响应时间：使交互用户的响应时间尽可能短

吞吐量：使单位时间内处理的进程数尽可能多

本文也将围绕这四个中心目标进行方案的设计。

1. **方案设计基本思想**

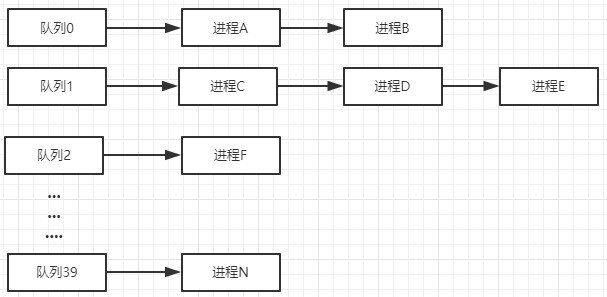
2.1 动态优先级的计算方案设计

进程的动态优先级是由进程的静态优先级和进程在系统中等待CPU的时间所决定的。进程在系统中等待CPU的时间越长，动态优先级会逐渐上升。进程在CPU运行的时间越长，动态优先级逐渐下降。传统的计算方法一般仅采用一次线性函数，例如：

但考虑到系统进程的优先级可能由用户进行自定义，为了使系统能够更加及时地做出响应，体现出实时系统的特点，我们增加了“进程紧急程度”这以概念。并将其分为3个等级（k=3:普通进程；k=6：很紧急）。同时，为了公平性，尽量使等待时间较长的进程尽快获得响应，同时将等待时间较短且立即被运行的进程相对“延后”，我们设计动态优先级计算方案如下：

2.2 就绪队列的设计

Linux中静态优先级范围是100~139，因此我们在设计等待进程的存储结构时，设计成一个多级（0~39级）队列。分别对应静态优先级100~139。在初始化就绪队列时，首先按照进程的静态优先级分别分配到每个队列当中。



就绪队列示意图

2.3 时间片的设计

调度策略必须规定有合适的时间片。时间片过长会导致系统对交互进程的相应欠佳，时间片太短会增大进程切换带来的处理器损耗。因此，我们考虑针对不同的就绪队列分别设置不同的时间片。针对紧急程度较高的，设置较长的时间片，而对优先级较低的队列分配较短的时间片。Linux的时间片范围在5ms~200ms，因此我们在设置时间片时，将最低有限队列的时间片设置为5ms，而最高的优先级队列设置为200ms。中间每级以5ms进行递增。

1. **算法详解**

***Step 1*** 系统根据待运行的多个进程同时产生相应的PCB控制块。根据系统对每个进程设置的静态优先级，将这些进程分配到各个就绪队列中。每个队列中进程无序排放，所有进程紧急程度权值初始化为3。

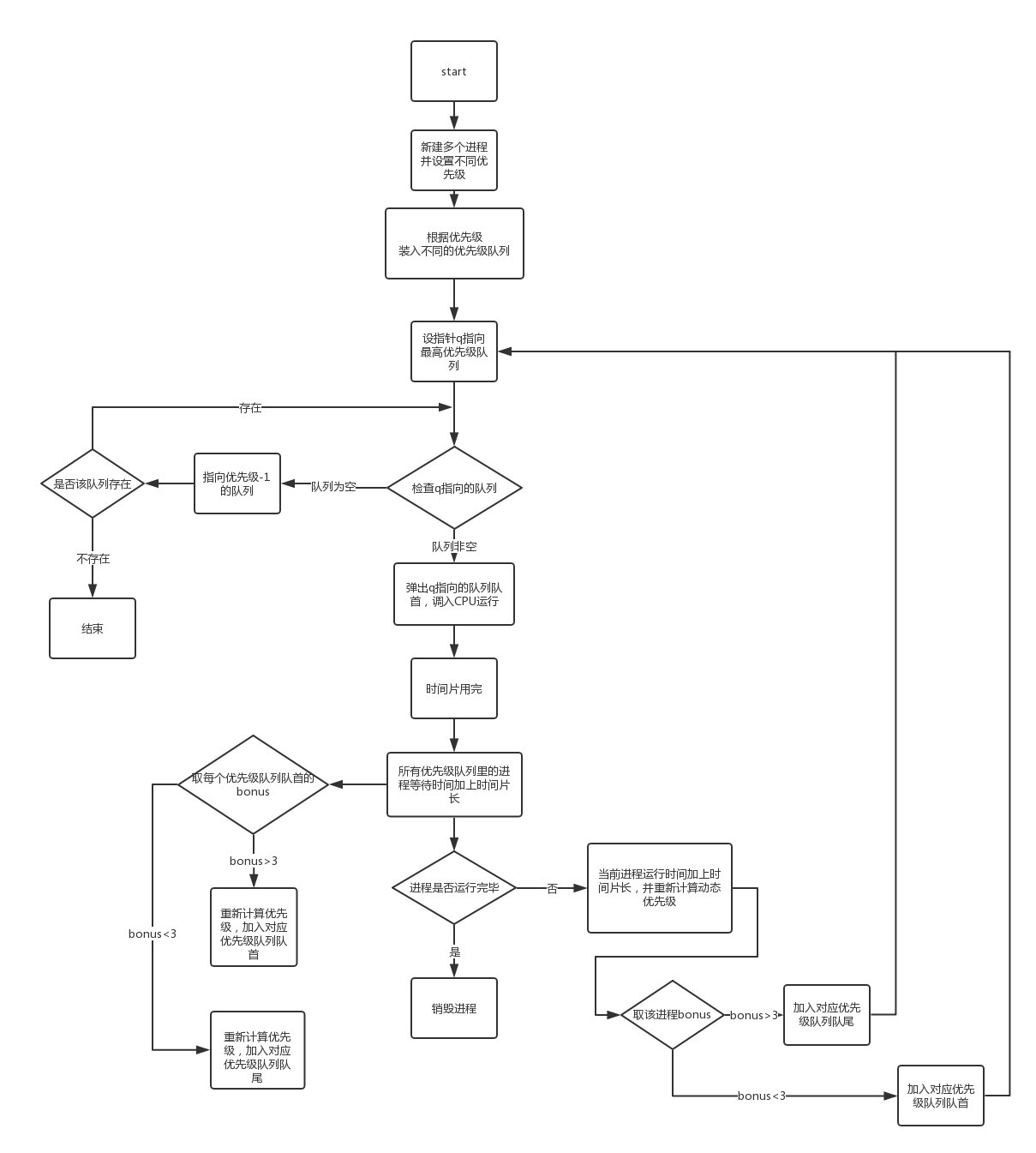
***Step 2***  依次扫描就绪队列，从第一个不为空的就绪队列开始，将队首进程投入到CPU中运行。

***Step 3*** 经过一个时间片之后，首先判断当前运行进程的是否已经运行完毕，若运行完毕，则直接抛去。若未结束，则运行时间增加一个时间片单位，而就绪队列中的所有进程的等待时间增加一个时间片单位。

***Step 4*** 重新计算40个就绪队列队首进程的动态优先级，并通过调度系统重新分配到不同的队列中。

***Step 5*** 调度系统首先判断该进程是否为紧急进程，若为紧急进程则将该进程添加到新就绪队列的队首，若为普通进程则将该进程添加到新就绪队列的队尾。返回***Step 2***

**算法流程图：**



1. **数据结构与伪码**

|  |
| --- |
| PCB设置：  C:\Users\FGX\Documents\Tencent Files\1007384211\Image\C2C\}(L6[}H[7V7Z8ZV}RFIRT_S.png |

|  |
| --- |
| 就绪队列结构：  C:\Users\FGX\Documents\Tencent Files\1007384211\Image\C2C\XKV9U`FI[EZU3WU9SKSHD)D.png |

|  |
| --- |
| 队列操作：  C:\Users\FGX\Documents\Tencent Files\1007384211\Image\C2C\}7~SZR~()JFCTFDW(~D(THU.png |

|  |
| --- |
| 进程函数操作：  struct task\* creat(char \*name, int static\_prio)  {  struct task \*p = (struct task\*)malloc(sizeof(struct task\*));  p->name = name;  p->pid = 分配pid;  p->static\_prio = static\_prio;  p->run\_time\_in\_cpu = 0;  p->wait\_time = 0;  p->prio = static\_prio; //初始动态优先级等于静态优先级  return p;  }  //进程调入CPU  void dispatch()  {  int i;  for (i = 0; i < 140; i++)  {  if (requeue[i] != NULL)  {  struct task \*p;  p = requeue[i].Dequeue(); //取队首  p->state = 1; //改为运行态  进程调入CPU运行  break;  }  }  }  //时间片到，等待进程处理  void wait(struct task\* p)  {  p->wait\_time += 时间片长;  p->prio += (p->run\_time\_in\_cpu) / 时间片长 + (p->bonus)(p->wait\_time) / 时间片长;  requeue[p->prio].Enqueue(p); //插入该优先级队列队尾  }  //时间片到，运行进程处理  void running(struct task\* p)  {  p->run\_time\_in\_cpu += 时间片长;  p->state=0 //转入等待状态  p->prio += (p->run\_time\_in\_cpu) / 时间片长 + (p->bonus)(p->wait\_time) / 时间片长;  if (进程运行完毕)  exit(p);  else  requeue[p->prio].Enqueue(p); //插入该优先级队列队尾  }  //进程销毁  void exit(struct task\* p)  {  free(p);  } |

1. **总结**

**方案优点：**

1. 本方案针对不同的进程进行不同程度的优先级的提高或减少，对于等待等待时间过长的，其优先级提升相对较快。而对于等待时间较短即被运行的进程，在运行过后，将其相对延后。
2. 本方案每次仅对40个就绪队列队首进程进行动态优先级的计算，既充分考虑减少“饥饿程度”，与所有进程全部重新计算优先级相比，又大大减少了计算量。
3. 本方案针对不同的优先级就绪队列设置不同的时间片，从而避免时间片统一情况下，造成的CPU利用率较低或切换进程的时间开销。

**方案缺点：**

1. 为了尽量降低计算开销，以至于空间开销较大。
2. 算法实现较为复杂
3. 只保证了优先级高的进程优先运行，但并不保证进程用了多少处理器时间。另外，未能充分考虑调度的公平性
4. **问题思考**

**如何统计进程的等待时间以及进程在CPU上的运行时间？**

等待时间：当进程进入就绪队列时开始计时，每经过一个时间片的时间，就为该进程的等待加上1个时间片，若进程从由就绪态转为运行态时，就清0。

运行时间：当进程进入CPU运行了一个时间片后，就为该进程的运行时间加上1个时间片。、

**时钟中断在进程调度中有什么作用？**

时钟中断的主要工作是处理和时间有关的所有信息、决定是否执行调度程序。和时间有关的所有信息包括系统时间、进程的时间片、延时、使用CPU 的时间、各种定时器，进程更新后的时间片为进程调度提供依据，然后在时钟中断返回时决定是否要执行调度程序。

系统越在极短时间内产生时钟中断，系统就越能对当前发生事件做出快速响应，系统的实时性越能提高。所以可提高系统时钟频率可改善系统实时性能。但是，当系统时钟中断频率增加时，为响应中断发生而要保存当前现场的工作量也大量增加，最后其负面作用就会超过改变时钟频率所带来的益处，使整个系统的性能下降。

1. **小组分工**

冯冠玺：文档撰写，算法设计，PPT演讲

陈鸿翔：算法设计，数据结构设计

金大伟：资料收集，参与讨论

1. **参考文献**

【1】贺炎, 刘曙霞. Linux的进程调度策略[J]. 电子科技, 2004(5):31-34.

【2】赵霞, 于重重. Linux进程调度策略的研究[J]. 微型电脑应用, 2001(1):20-22.

【3】王刚, 余兆, 李晓勇,等. Linux 2.6内核进程调度分析[J]. 计算机应用与软件, 2007, 24(9):162-164.

【4】杨静, 李炜, 万峰松,等. Linux内核2.6进程调度分析与改进[J]. 计算机技术与发展, 2009, 19(7):105-107.

【5】邹治锋. 基于Linux的进程调度算法的改进与实现[D]. 江南大学, 2006.

【6】轮转调度算法和动态优先级调度算法

<http://blog.csdn.net/qzq2514/article/details/53590600>

【7】 Linux源码分析与研究

<http://www.docin.com/p-947237633-f3.html>