Linux 内核 2.6.24 的 CFS 调度器分析

杜慧江 王云光
(上海医疗器械高等专科学校医用电子仪器系 上海 200093)

摘 要 Linux 内核 2.6.24 采用了新的调度器 CFS(Completely Fair Scheduler),以公平的原则对待所有任务。其性能和对交互进程的响应速度都超过之前版本内核的 O(1)调度器,并且算法与实现也更简单。先简单对比了 O(1)和 CFS,然后结合源代码对 CFS 的算法和实现作了分析。

关键词 Linux 内核 调度器 CFS

ANALYSIS OF COMPLETELY FAIR SCHEDULER IN LINUX KERNEL 2.6.24

Du Huijiang Wang Yunguang
(*Department of Medical Electronics Apparatus, Shanghai Medical Instrumentation College, Shanghai 200093, China*)

Abstract Linux kernel 2.6.24 has adopted a new Completely Fair Scheduler, the principle of it is the fair treatment on all tasks. Its performance and responding speed on interactive tasks are all superior to previous scheduler O(1), and its algorithm and implementation are simpler too. In this paper we draw a simple comparison between CFS and O(1), and then briefly analyse the algorithm and implementation of CFS according to its source code.

Keywords Linux Kernel Scheduler CFS

0 引 言

Linux 的稳定内核版本从 2.6.24 开始,使用了新的调度程序 CFS,能够更加公平高效地调度所有任务(包括进程和线程),平衡交互任务和普通任务。本文首先将 2.6.22 内核的 O(1)调度器与 CFS 进行比较,然后根据源代码对 CFS 的实现进行分析。

1 O(1)调度器

Linux2.6.22 内核使用的是 O(1)调度器,它支持 SMP 并且能保证无论系统的负载和处理器的数目如何,用来选择合适的任务并且给它分配处理器所花费的时间是固定的。

O(1)调度器的任务调度主要包括两部分工作:

(1) *动态计算任务优先级* 普通任务的优先级是动态计算的,由公式 dynamic priority = max(100, min(static priority-bonus + 5,139))[2] 得出。其中 bonus 取决于任务的平均睡眠时间。普通任务的平均睡眠时间越长,其 bonus 就越大,从而得到更高的优先级。静态优先级越高,任务所能分配到的时间片也越长。实时任务的优先级不会动态修改,而且总是比普通任务的优先级高。

(2) *选出下一个获得 CPU 的任务* 普通任务的调度选择算法基于任务的优先级,当前拥有最高优先级的任务被调度器选中。调度器为每个 CPU 维护了两个任务队列数组:active 数组和 expired 数组。数组中的元素保存着某一优先级的任务队列指针。当需要选择下一个任务时,不用遍历所有任务队列,而是直接从 active 数组中选择当前最高优先级队列中的第一个任务。这种算法的复杂度为 O(1)。

为提高交互式任务的响应速度,每次时钟中断中,当前任务的时间片被减 1,当时间片为 0 时,调度器判断其类型,如果是交互式任务,则重置其时间片并重新插入 active 数组;否则将其从 active 数组中移到 expired 数组,这样交互式任务就总能优先获得 CPU。但是这些任务不能一直留在 active 数组中,当交互任务已经占用 CPU 的时间超过一个固定值后,也会被移到 expired 数组中。若 active 数组为空,则交换 active 数组和 expired 数组,开始新的一轮调度。

O(1)调度器算法的不足之处在于会在很多情况下失效,导致交互式任务反应缓慢。

2 CFS 调度器

CFS 调度器由 Ingo Molnar 提出,采用了完全公平的思想对待所有任务。但是"公平"不等于同等对待所有任务,而是指相对长程的、统计上的公平。在每个小的时间区间很可能看起来并不公平,原因可能是需要对以往的不公平作出补偿、系统的负载变化等。

CFS 还增加了一个调度器模块管理器,各种不同的调度算法可以作为一个模块注册到该管理器中。不同的任务可以选择使用不同的调度器模块,例如实时任务可以选择实时任务调度模块。Linux 2.6.24 内核中,CFS 实现了两个调度算法,CFS 算法模块和实时调度模块。对应实时任务,将使用实时调度模块;对应普通任务则使用 CFS 算法。

CFS 的算法和实现都更简单,现有的测试表明其性能也比

收稿日期:2008-09-09。杜慧江,讲师,主研领域:软件与算法。

O(1)更好。作者公布的数据表明 CFS rc6-devel 的性能比 O(1)提高 3.1%;二进制文件的尺寸 SMP 版 CFS 比 O(1)小 8.5%[3]。

2.1 算　法

CFS 弃用了 active/expired 数组和动态计算优先级,不跟踪任务的睡眠时间,也不区别是否交互任务,而是用基于时间计算键值的红黑树来选取下一个任务[4];也不再分配时间片,而是根据所有任务占用 CPU 时间的状态来调度任务。所有就绪的任务都依据键值的大小被插入红黑树的叶子节点。键值越小越靠左,键值越大越靠右。在每个调度点,调度器都会选择红黑树中最左边的叶子节点作为下一个将获得 CPU 的任务,该选择操作的时间复杂度是 O(log2N)。

在每个时钟中断里,首先更新调度信息,然后调整当前任务在红黑树中的位置。如果发现当前任务不再是最左边的叶子节点,就标记 need_resched 标志,中断返回时就会调用 scheduler_tick()完成任务切换;否则当前任务继续占用 CPU。

2.2 键值计算

键值由三个因子计算得出:一是任务已经占用的 CPU 时间;二是当前任务的 nice 值;三是当前的 CPU 负载。

当前任务已占用的 CPU 时间对其键值的影响最大,该值越大,键值就越大。兼顾了处理器上每个任务的权值和处理器的负载情况,计时使用的并不是实际的时钟,而是虚拟时钟。它的前进步伐与该处理器上的任务权重成反比。当系统内有多个任务时,虚拟时钟的步调就按总权重大小成比例地慢下来,即虚拟时间单元会按比例变长。每个任务也都有自己的虚拟时钟,它们的前进步伐与自己的权重成反比。

nice 是一个可以调整的值,它提供给任务修改运行优先权的程序能力。-20~0 代表高优先级,0~19 代表低优先级。高 nice 值导致低的运行级别,低 nice 值导致高的运行级别。所有任务默认的 nice 值是 0。运行的任务可以提高自己的 nice 值,但是只有以 root 运行的任务可以降低 nice。因此 nice 值越大,键值也越大。

任务已经占用的 CPU 时间用变量 fair_clock 表示,另一个变量是 wait_runtime,表示当前任务已经等待的时间。任务插入红黑树的键值为 fair_clock - wait_runtime + nice。键值代表了一个任务的公平程度。该值越大,越应该让出 CPU。

对于交互式任务,运行时间少而等待时间长,因此它能拥有很小的红黑树键值,更靠近红黑树的左边,从而可以优先得到响应。

2.3 源代码分析

我们参考内核 2.6.24 源代码中的 sched.c 和 sched_fair.c 来分析:

sched.c 中的 scheduler_tick()函数会被时钟中断直接调用,它先更新当前 runqueue 的 clock 值和时间戳,然后刷新 CPU 负载情况,调用当前任务的 sched_class 类的 task_tick()函数。在 sched_class 类的定义中,task_tick() = task_tick_fair()。task_tick_fair()在 sched_fair.c 中:

```
static void task_tick_fair(struct rq * rq,struct task_struct * curr)
{
  struct cfs_rq * cfs_rq;
  struct sched_entity * se = &curr -> se;
  for_each_sched_entity(se) {
    cfs_rq = cfs_rq_of(se);
    entity_tick(cfs_rq,se);
  }
}
```

这个函数是让当前的待调度任务执行 entity_tick()函数:

```
static void entity_tick(struct cfs_rq * cfs_rq,struct sched_entity * curr)
{
  struct sched_entity * next;
    dequeue_entity(cfs_rq,curr,0);
    enqueue_entity(cfs_rq,curr,0);
    next = __pick_next_entity(cfs_rq);
    if (next == curr)
      return;
    __check_preempt_curr_fair(cfs_rq,next,curr,sched_granularity(cfs_rq));
}
```

通过调用 dequeue_entity()和 enqueue_entity(),将任务从红黑树中删除再插入来调整任务在红黑树中的位置。_pick_next_entity()返回红黑树中最左边的叶子节点,如果此节点不是当前任务,就调用_check_preempt_curr_fair()设置调度标志,当中断返回时就会调用 schedule_tick()进行调度,替换当前任务。enqueue_entity()的源码如下:

```
static void enqueue_entity(struct cfs_rq * cfs_rq,struct sched_entity * se,int wakeup)
{
  update_curr(cfs_rq);
  if (wakeup)
      enqueue_sleeper(cfs_rq,se);
  update_stats_enqueue(cfs_rq,se);
  __enqueue_entity(cfs_rq,se);
}
```

它首先更新调度信息,然后将任务插入红黑树中。其中 update_curr()完成调度信息的更新,dequeue_entity()负责将当前任务从红黑树中删除。update_curr()源代码如下:

```
static void update_curr(struct cfs_rq * cfs_rq)
{
  struct sched_entity * curr = cfs_rq_curr(cfs_rq);
  unsigned long delta_exec;
  if (unlikely(! curr))
      return;
  delta_exec = (unsigned long)(rq_of(cfs_rq) -> clock-curr -> exec_start);
  curr -> delta_exec += delta_exec;
  if (unlikely(curr -> delta_exec > sysctl_sched_stat_granularity)) {
      __update_curr(cfs_rq,curr);
      curr -> delta_exec = 0;
  }
  curr - > exec_start = rq_of(cfs_rq) - > clock;
}
```

此函数首先统计当前任务所获得的 CPU 时间,rq_of(cfs_rq) -> clock 值在时钟中断中被更新,curr -> exec_start 是当前任务开始获得 CPU 时的时间戳。两值相减就是当前任务所获得的 CPU 时间。将该变量存入 curr -> delta_exec 中,然后调用__update_curr()函数。

```
static inline void __update_curr(struct cfs_rq * cfs_rq,struct sched_entity * curr)
```

```
{
    unsigned long delta,delta_exec,delta_fair,delta_mine;
    struct load_weight *lw = &cfs_rq->load;
    unsigned long load = lw->weight;
    delta_exec = curr->delta_exec;
    schedstat_set(curr->exec_max,max((u64)delta_exec,curr->exec_
max));
    curr->sum_exec_runtime += delta_exec;
    cfs_rq->exec_clock += delta_exec;
    if(unlikely(!load))
        return;
    delta_fair = calc_delta_fair(delta_exec,lw);
    delta_mine = calc_delta_mine(delta_exec,curr->load.weight,lw);
    if (cfs_rq->sleeper_bonus > sysctl_sched_min_granularity) {
        delta = min((u64)delta_mine,cfs_rq->sleeper_bonus);
        delta = min(delta,(unsigned long)(
          (long)sysctl_sched_runtime_limit-curr->wait_runtime));
        cfs_rq->sleeper_bonus -= delta;
        delta_mine -= delta;
    }
    cfs_rq->fair_clock += delta_fair;
    add_wait_runtime(cfs_rq,curr,delta_mine-delta_exec);
}
```

__update_curr()更新 fair_clock 和 wait_runtime。如果当前任务队列任务个数发生改变,CPU 负载也会改变,就需要结合负载的情况调整 fair_clock 和 wait_runtime。delta_exec 保存了前面获得的当前任务所占用的 CPU 时间。calc_delta_fair()根据 cpu 负载对 delta_exec 进行修正,然后将结果保存到 delta_fair 中,最后将 fair_clock 增加 delta_fair。calc_delta_mine()根据 nice 值(保存在 curr->load.weight 中)和 cpu 负载修正 delta_exec,将结果保存在 delta_mine 中。delta_mine 就表示当前任务应该获得的 CPU 时间。

修正的最终结果可以近似看成键值 fair_clock - wait_runtime + nice 增加了一倍的 delta_exec 值。键值增加,使得当前任务在红黑树中可能向右移动而失去坐左边叶子节点的位置。

此外,还有 yield_task_fair()函数,当某个任务自愿放弃 CPU 时,对当前任务进行先删除再插入操作来调整红黑树。

将上面几个函数画出调用关系,如图 1 所示。

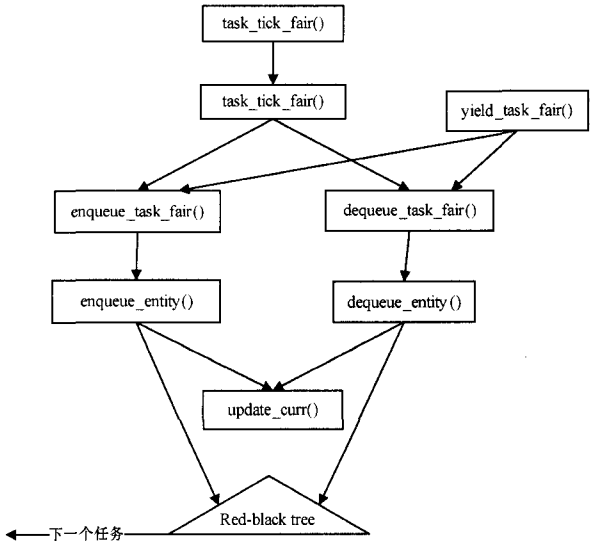

图 1 各函数间调用关系

3 结束语

CFS 的中心工作就是根据任务已经占用的 CPU 时间、当前任务的 nice 值和当前的 CPU 负载这三个因子,动态地计算键值并调整红黑树,红黑树中最左边的叶子节点就是下一个获得 CPU 的任务。该算法降低了调度计算的复杂度,兼顾了效率和公平;引入了模块管理器,可以加载不同的调度模块,灵活性、扩展性更好。

参 考 文 献

[1] Linux 内核源代码. http://www.kernel.org.

[2] BOVET D,CESATI M. Understanding the Linux Kernel[M]. O'Reilly Media,Inc,2005:260.

[3] MOLNAR Ingo. CFS-devel, performance improvements. http://lkml.org/lkml/2007/9/11/395.

[4] MOLNAR Ingo. Modular Scheduler Core and Completely Fair Scheduler. http://lkml.org/lkml/2007/4/13/180.

[5] 刘明. Linux 调度器发展简述. http://www.ibm.com/developerworks/cn/linux/l-cn-scheduler/index.html.

(上接第 80 页)

从图 2 中可以看出,嵌入秘密信息后音频信号在采样点 81667 - 81866 区间的波形幅度仅有微小的变化,较好地保持了原有波形,透明性比较好。根据 HAS 的掩蔽特性,人耳不会察觉这么细微的变化。

3 结束语

实验证明,这种基于 DCT 变换的数字音频密写算法具有较大的嵌入量(密写容量最高可达到 S/4,S 为音频信号的采样点)、较好的透明性、较强的健壮性,能经受重新量化、重采样、添加噪声、多种低通滤波、音频格式转换等常见信号处理及攻击,且提取秘密信息属盲提取。由于载体是音频信号,而隐藏的秘密信息是图像,完全掩盖了秘密信息的存在。本密写算法适合在互联网等公共信道传输秘密信息。

参 考 文 献

[1] Bender W,et al. Techniques for data hiding[J]. IBM System Journal,1996,35(3/4):313 - 336.

[2] Gruhl D,Lu A,Bender W. Echo hiding Information hiding[C]//First international workshop,Cambridge,1996,1174:295 - 315.

[3] Wang Ye. A new watermarking method of digital audio content for copyright protection[C]//Proceedings of ICSP'98,1998,1:1420 - 1423.

[4] 钮心忻,杨义先. 基于小波变换的数字水印隐藏与检测算法[J]. 计算机学报,2000,23(1):21 - 27.

[5] 李跃强,孙星明. 一种用音频作载体的信息隐藏算法[J]. 计算机应用研究,2006,23(5):29 - 33.

[6] 柴毅,刘一均,郭茂耘,等. 鲁棒的高容量音频信息隐藏算法[J]. 计算机工程与应用,2008,34(4):162 - 166.

[7] 李跃强,孙星明. 基于倒谱变换的数字音频健壮盲水印算法[J]. 计算机工程与应用,2005,41(35):52 - 55.