3.2 CFS 调度器

在阅读本章之前请思考关于 Linux 内核普通进程调度的几个小问题：

1. *请简述你对进程调度器的理解，早期 Linux 内核调度器包括 O(N)和 O(1)调度器是如何工作的？*
2. *进程优先级、nice 以及权重之间有啥关系？*
3. *请简述 CFS 调度器是如何工作的？*
4. *CFS 调度器中 vruntime 是如何计算的？*
5. *vruntime 是什么时候更新的？*
6. *CFS 调度器里 min_vruntime 有什么作用？*
7. *CFS 调度器对新创建的进程以及刚唤醒的进程有何关照？*
8. *普通进程的平均负载 load_avg_contrib 是如何计算的？runnable_avg_sum 和 runnable_avg_period 是什么含义？*
9. *内核代码中定义了好几个表，请说出它们的含义。prio_to_weight、prio_to_wmult、runnable_avg_yN_inv、runnable_avg_yN_sum。*
10. *如果一个普通进程在就绪队列里等待了很长时间才被调度，那么它的平均负载又该如何计算？*

Linux 内核作为一个通用操作系统，需要兼顾各种各样类型的进程，包括实时进程、交互式进程、批处理进程等。每种类型进程都有其特别的行为特征。

- 交互式进程：与人机交互的进程，和鼠标、键盘、触摸屏等相关的应用比如 vim 编辑器等，它们一直在睡眠等待用户召唤它们。这类进程的特点是希望系统响应时间快，否则用户就会抱怨系统卡顿。
- 批处理进程：此类进程默默工作和付出，可能会占用比较多的系统资源，比如编译代码等。
- 实时进程：有些应用对整体延时有严格要求，比如现在很火的 VR 设备，从头部转动到视频显示需要控制到 19ms 以内否则人会出现眩晕。有些工业控制系统，稍微的延迟会导致严重的事故发生。

不同的进程采用不同的调度策略，目前 Linux 内核中默认实现了 4 种调度策略，分别是 deadline、realtime、CFS 和 idle，它们分别使用 struct sched_class 来定义成调度类。

本章主要是讲述普通进程的调度，包括交互进程和批处理进程等。在 CFS 调度器出现之前，早期 Linux 内核中曾经出现过两个调度器，分别是 O(N)和 O(1)调度器。O(N)调度器就是 Linus 在 1992 年发布的那款调度器。该调度器算法比较简洁，从就绪队列中所有进程的优先级进行比较，然后选择一个最高优先级的进程作为下一个调度进程。每个进程有一个固定时间片，当进程时间片使用完之后调度器会选择下一个调度进程，当所有进程都运行一遍之后再重新分配时间片。这个调度器选择下一个调度进程需要遍历整个就绪队列，花费 O(N)时间。

在 Linux2.6.23 内核之前有一款名为 O(1)的调度器，优化了选择下一个进程的时间。它为每个 CPU 维护一组进程优先级队列，每个优先级一个队列，这样在选择下一个进程的时候，只需要查询优先级队列相应的位图就可以知道那个队列中有就绪进程，所以这个查询时间为常数 O(1)。

O(1)调度器在处理某些交互式进程时依然有不少问题，特别是有一些测试场景下导致交互式进程反应缓慢，另外对 NUMA 支持也不完善，因此大量难以维护和阅读的代码被加入到该调

度器中。所以 Linux 内核社区的一位传奇人物 Con Kolivas[①]提出了 RSDL（楼梯调度算法）来实现公平性，在社区的一番争斗之后，红帽子的 Ingo Molnar 在借鉴 RSDL 的思想憋出一个 CFS 调度算法。

这 4 种调度类通过 next 指针串联在一起。用户空间程序可以使用调度策略 API 函数（sched_setscheduler()[②]）来设定用户进程的调度策略。其中 SCHED_NORMAL 和 SCHED_BATCH 使用 CFS 调度器，SCHED_FIFO 和 SCHED_RR 使用 realtime 调度器，SCHED_IDLE 指的是 idle 调度，SCHED_DEADLINE 指的是 deadline 调度器。

[include/uapi/linux/sched.h]

```
/*
 * Scheduling policies
 */
#define SCHED_NORMAL        0
#define SCHED_FIFO      1
#define SCHED_RR        2
#define SCHED_BATCH         3
/* SCHED_ISO: reserved but not implemented yet */
#define SCHED_IDLE      5
#define SCHED_DEADLINE      6
```

3.2.1 权重计算

内核使用[0~139]的数值来表示进程的优先级，数值越低优先级越高。优先级 0~99 是给实时进程使用，100~139 是给普通进程使用。另外在用户空间有一个传统的变量 nice 值来映射到普通进程的优先级即[100~139]。

进程 PCB 描述符 struct task_struct 数据结构中有 3 个成员来描述进程的优先级。

```
struct task_struct {
    ...
    int prio;
    int static_prio;
    int normal_prio;
    unsigned int rt_priority;
    ...
};
```

static_prio 是静态优先级，在进程启动时分配的优先级。内核不存储 nice 值取而代之的是 static_prio。内核有一个宏 NICE_TO_PRIO()实现由 nice 值转换成 static_prio。它之所以被称为静态优先级是因为它不会随着时间而改变，用户可以通过 nice 或者 sched_setscheduler 等系统调用来修改该值。normal_prio 是基于 static_prio 和调度策略计算出来的优先级，进程在创建时会继承父进程的 normal_prio。对于普通进程来说，normal_prio 等同于 static_prio，对于实时进程，会根据 rt_priority 重新计算 normal_prio，见 effective_prio()函数。prio 保存着进程的动态优

① Con Kolivas 是内核传奇的开发者，他主业是麻醉师，在内核社区中一直关注用户体验的提升，设计了相当不错的调度器算法，最终没有被社区采纳。后来他有设计了一款名为 BFS 的调度器。

② sched_setscheduler(), sched_getscheduler() – 用户空间程序系统调用 API 用来设置和获取内核调度器的调度策略和参数。

先级，是调度类考虑的优先级，有些情况下需要暂时提高进程优先级，比如实时互斥量等。rt_priority 是实时进程的优先级。

内核使用一个 struct load_weight 数据结构来记录调度实体的权重信息（weight）。

```
struct load_weight {
    unsigned long weight;
    u32 inv_weight;
};
```

其中 weight 是调度实体的权重，inv_weight 是 inverse weight 的缩写，它是权重的一个中间计算结果，如何使用稍后会看到。调度实体的数据结构中已经内嵌了 struct load_weight 结构体来描述调度实体的权重。

```
struct sched_entity {
    struct load_weight load;        /* for load-balancing */
...
}
```

因此代码里经常通过 p->se.load 来获取进程 p 的权重信息。刚说了 nice 值的范围是从-20~19，进程默认的 nice 值为 0。这些值含义类似级别，可以理解成有 40 个等级，nice 值越高则优先级越低，反之亦然。例如一个 CPU 密集型的应用程序 nice 值从 0 增加到 1，那么它相对于其他 nice 值为 0 的应用程序将减少 10%的 CPU 时间。因此进程每减低一个 nice 级别，优先级则提高一个级别，相应的进程则多获得 10%的 CPU 时间，反之每提升一个 nice 级别，优先级则降低一个级别，相应的进程则少 10%的 CPU 时间。内核为了计算方便约定 nice 值为 0 的权重值为 1024，其他 nice 值对应的权重值则通过查表的方式[①]来获取，内核预先计算好了一个表 prio_to_weight[40]表下标对应 nice 值[-20~19]。

[kernel/sched/sched.h]

```
/*
 * Nice levels are multiplicative, with a gentle 10% change for every
 * nice level changed. I.e. when a CPU-bound task goes from nice 0 to
 * nice 1, it will get ~10% less CPU time than another CPU-bound task
 * that remained on nice 0.
 *
 * The "10% effect" is relative and cumulative: from _any_ nice level,
 * if you go up 1 level, it's -10% CPU usage, if you go down 1 level
 * it's +10% CPU usage. (to achieve that we use a multiplier of 1.25.
 * If a task goes up by ~10% and another task goes down by ~10% then
 * the relative distance between them is ~25%.)
 */
static const int prio_to_weight[40] = {
 /* -20 */     88761,     71755,     56483,     46273,     36291,
 /* -15 */     29154,     23254,     18705,     14949,     11916,
 /* -10 */      9548,      7620,      6100,      4904,      3906,
 /*  -5 */      3121,      2501,      1991,      1586,      1277,
 /*   0 */      1024,       820,       655,       526,       423,
 /*   5 */       335,       272,       215,       172,       137,
 /*  10 */       110,        87,        70,        56,        45,
```

[①] 查表的方式是最快和粗鲁的一种优化方法，比如写一个函数来计算 prio_to_weight 永远也没有查表来的快。再比如程序中需要用到 100 以内的质数，预先定义好一个 100 以内质数表，查表的方式比用一个函数的方式要快很多。内核常常喜欢这种优化方法。

```
 /*  15 */        36,        29,        23,        18,        15,
};
```

刚才说的 10%的影响是相对及累加的，比如一个进程增加了 10%的 CPU 时间则另外一个进程减少了 10%，那么直接的差距大约是 20%，因此这里使用一个系数 1.25 来计算的。举个例子，进程 A 和进程 B 都是 nice 值为 0，那权重值都是 1024，它们获得 CPU 的时间都是 50%，计算公式 1024/(1024+1024)=50%。假设进程 A 增加一个 nice 值，即 nice=1，进程 B 的 nice 值不变，那么这时候进程 B 应该获得 55%的 CPU 时间，进程 A 应该是 45%才对。我们利用 prio_to_weight[]这个表来计算一下，进程 A = 820/(1024+820) = 45%，而进程 B = 1024/(1024+820) = 55%，注意这里是近似等于。

内核中还提供另外一个表 prio_to_wmult[40]，它也是预先计算好的。

[kernel/sched/sched.h]

```
/*
 * Inverse (2^32/x) values of the prio_to_weight[] array, precalculated.
 *
 * In cases where the weight does not change often, we can use the
 * precalculated inverse to speed up arithmetics by turning divisions
 * into multiplications:
 */
static const u32 prio_to_wmult[40] = {
 /* -20 */     48388,     59856,     76040,     92818,    118348,
 /* -15 */    147320,    184698,    229616,    287308,    360437,
 /* -10 */    449829,    563644,    704093,    875809,   1099582,
 /*  -5 */   1376151,   1717300,   2157191,   2708050,   3363326,
 /*   0 */   4194304,   5237765,   6557202,   8165337,  10153587,
 /*   5 */  12820798,  15790321,  19976592,  24970740,  31350126,
 /*  10 */  39045157,  49367440,  61356676,  76695844,  95443717,
 /*  15 */ 119304647, 148102320, 186737708, 238609294, 286331153,
};
```

prio_to_wmult[]表的计算公式如下：

$$inv\_weight = \frac{2^{32}}{weight}$$

其中 inv_weight 是 inverse weight 的缩写，指权重被倒转了，这个主要是为后面计算方便。

内核提供一个函数来查询这两个表然后把值存放在 p->se.load 数据结构中，即 struct load_weight 结构中。

```
static void set_load_weight(struct task_struct *p)
{
     int prio = p->static_prio - MAX_RT_PRIO;
     struct load_weight *load = &p->se.load;

     load->weight = scale_load(prio_to_weight[prio]);
     load->inv_weight = prio_to_wmult[prio];
}
```

设计这个 prio_to_wmult[]表究竟做什么用途呢？

在 CFS 调度器中有一个计算虚拟时间的核心函数 calc_delta_fair()，它的计算公式是：

$$vruntime = \frac{delta\_exec * nice\_0\_weight}{weight}$$

其中 vruntime 表示进程虚拟的运行时间，delta_exec 则表示实际运行时间，nice_0_weight 表示 nice 为 0 的权重值，weight 表示该进程的权重值。

vruntime 该如何理解呢？

假设系统中只有三个进程 A、B 和 C，它们的 NICE 值都是 0，也就是权重值都是 1024。它们分配到的运行时间是一样的，即都应该分配到 1/3 的运行时间。那如果 ABC 三个进程的权重值不一样呢？

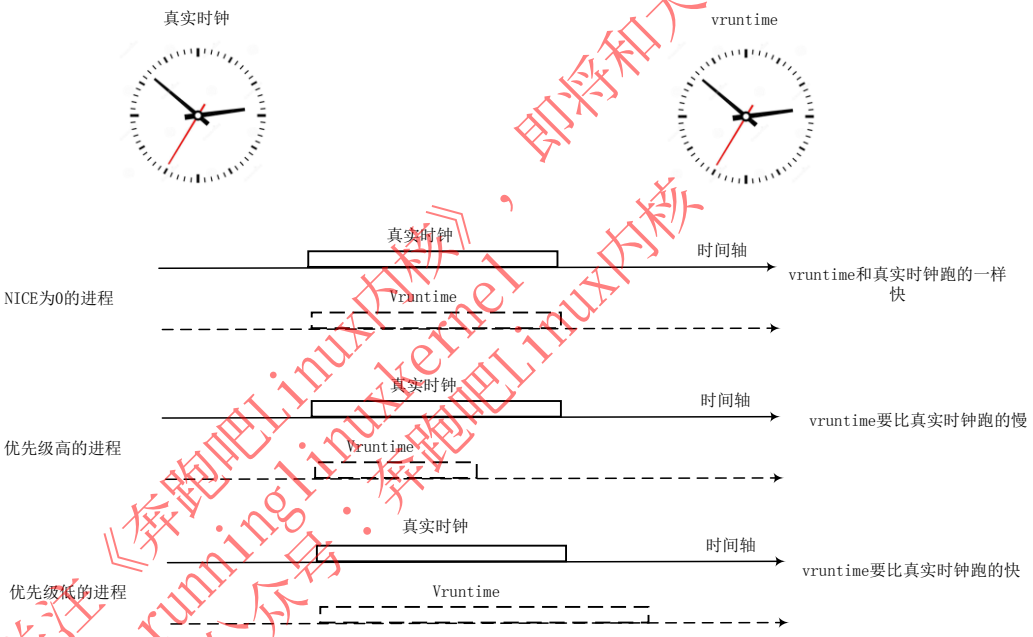

图 3.4 vruntime 和真实时钟对比

CFS 调度器抛弃以前固定时间片和固定调度周期的算法而是采用进程权重值的比重来计算实际运行时间。另外引入虚拟时钟的概念，每个进程的虚拟时间是实际应该运行时间相对于 NICE 值为 0 的权重的一个比例值。进程在按照各自不同的速率比在物理时钟节拍内前进。NICE 值小的进程，优先级高权重大，其虚拟时钟比真实时钟跑的慢，但是可以获得比较多的运行时间；反之 NICE 值大的进程，优先级低权重也低，其虚拟时钟比真实时钟跑的快反而获得比较少的运行时间。CFS 调度器总是选择虚拟时钟跑的慢的进程。CFS 调度器像一个多级变速箱，NICE 为 0 的进程是基准齿轮，其他各个进程在不同的变速比下相互追赶，从而达到公正公平。

假设某个进程 nice 值为 1，其权重值为 820，delta_exec=10ms，导入公式计算 vrumtime = (10*1024)/820，这里要涉及到浮点运算了。为了计算高效，函数 calc_delta_fair()的计算方式变成乘法和移位运行。