

CFS调度器

注: 本文中的所有图片均为组员绘制。

1 CFS调度器概述

Linux 进程调度算法经历了以下几个版本的发展：

- Round Robin调度器。(2.6之前)
- O(1) 调度器。(2.6.23之前)
- CFS 调度器。(2.6.23以及之后)

CFS调度器的要义在于，调度器总是从就绪队列中挑选虚拟运行时间(一个调度属性)最小的进程，也就是要维护一个优先队列。

尽管目前的YuanShen OS功能比较简单，原先单纯Round Robin调度器足以以非常低的开销完成调度任务(在小任务规模上红黑树显然没有什么优势，甚至是劣势)。不过作为学习者而言，手写一份类linux的CFS调度器应当是富有学习价值的。

我们最开始直接在原先数组扫描的基础上实现CFS调度算法，后来我们又换成了红黑树(不然不够炫酷)，在github仓库的提交记录里可以看到两个版本：

feature/CFS_Scheduler (红黑树实现)
feature/add_scheduler (数组实现)

最终的代码保存在目录./kernel/CFS/cfs.c

2 CFS调度算法

以下内容参考论文 *Completely Fair Scheduler and its tuning*：

在CFS调度算法中，每个进程被赋予一个vruntime(虚拟运行时间)，它通过权衡进程的实际运行时间和进程的权重而被计算：

$$\Delta \text{vruntime} = \frac{\Delta \text{runtime}}{\text{weight}}$$

weight是进程的调度权重，默认值为1024。这个值是怎么来的？事实上我们之后会为用户态进程提供一个set_priority的系统调用，基于它用户可以设置进程的"nice"，它和weight有这样的联系：

$$\text{weight} = 1024 \times 1.25^{-\text{nice}}$$

nice每调高1,理想情况下进程就会在原来的基础上少约10%的运行时间。CFS调度算法要求调度器每次选取vruntime最小的就绪进程，为它分配时间片，有公式：

$$\text{timeSlice} = \text{period} \times \frac{\text{weight}}{\text{total weight}}$$

其中period是调度器调度所有进程一轮的用时，权重越大分配到的时间片也越大。我们依照CFS调度算法的设计，确定

$$\text{period} = \max(\text{default latency}, n \times \text{min_granularity})$$

n 为就绪队列中的进程数量。换句话说，CFS调度器要使用一种灵活的方案。当时间富裕的时候，每个进程每隔period都可以分到一个时间片。当任务很多的时候，那么这个period也会随着变长，一个进程可能要等待更久的时间。

最后我们指出进程抢占发生的条件。当进程被时钟中断时，它不一定要出让CPU，比如它的时间片未必已经用完。min_granularity则是每个进程在调度后至少应该执行的时间，以免因为它优先级低而完全得不到时间片。综合考量，我们给出的条件是

$$\Delta \text{runtime} \geq \text{min_granularity} \quad , \quad \Delta \text{runtime} \geq \text{timeSlice} \quad \text{or} \quad \text{vruntime} \geq \text{min_vruntime}$$

也就是说运行时间超过最小时间粒度是必须要满足的，剩下两个条件满足一个即可。

3 CFS调度所需的数据结构

3.1 进程结构体的调度属性

首先往进程结构体内添加如下字段表示调度属性：

```
//struct proc
int vruntime;
int curruntime; //用微秒计数
int timeSlice;
int nice;
int weight;
```

vruntime在每次进程重新进入runnable的时候重新计算，公式为

$$\Delta \text{vruntime} = \frac{\text{curruntime} * 1024}{\text{weight}}$$

其中1024为nice值为默认值0时对应的权重，计算完后curruntime清零，表示这一轮调度内它分到的时间。至于时间片大小timeSlice，它在进程从就绪队列中被取出时才被计算

$$\text{timeSlice} = \text{period} \frac{\text{weight}}{\text{totalWeight}}$$

总权重是由RBTree类维护的，意指就绪队列中所有进程的权重之和。由于进程在就绪状态下调度属性不会发生改变，我们可以等到进程入队或者出队的时候才更新它。

3.2 基于红黑树的priorityQueue

红黑树是一种自平衡的二叉搜索树，能够在对数时间内进行插入、删除和查找操作。这使得 CFS 能够动态地调整进程的调度优先级，保持调度队列的高效性。

ready queue for CFS Scheduler

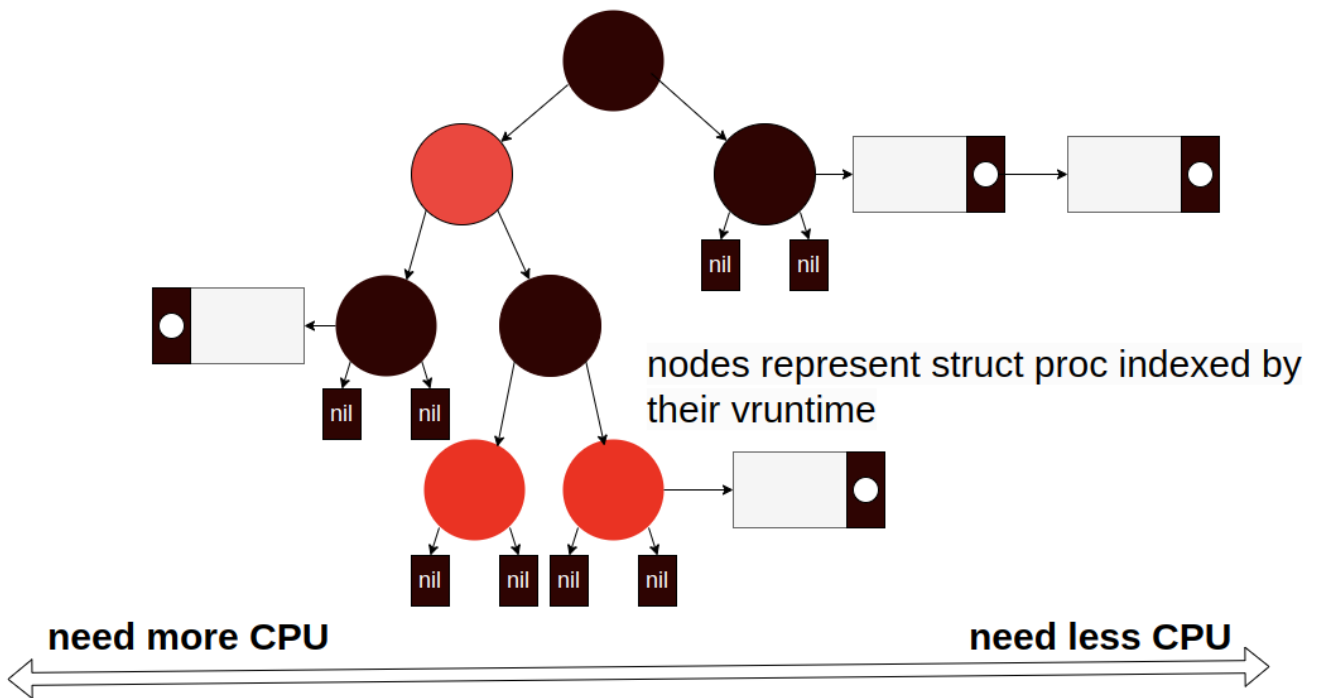


Fig1. YuanShen OS的CFS调度器原理(组员绘制)

如Fig1所示，我们根据vruntime，将所有的进程组织在一个由红黑树实现的优先队列中。当进程变为runnable时(例如被kill，wakeup或者fork时)，我们可以把进程插入队列的合适位置。而调度器永远是取下队头的那个进程。

为了组建红黑树，我们为进程结构体中加入了如下代码:

```
//struct proc
struct proc *left;
struct proc *right;
struct proc *rb_parent;
struct proc *next;
procColor color;
```

为什么会多了一个next？

加入next的原因是传统的红黑树不支持duplicate key，但是两个进程有相同的vruntime却是有可能的。显然它们都必须在rdy_queue内。

因此我们使用next在红黑树的节点处又维护一个链表，上面安置的进程具有相同的vruntime，有点像hashtable处理重复元素的方法，不过依旧有效。

最后，我们用RBTree类维护就绪队列的各种元信息,代码如下所示:

```
//cfs.h
extern const int min_granularity;
extern const int latency;
```

```

struct RBTREE {
    int count;
    int rbTreeWeight;
    struct proc *root;
    struct proc *min_vruntime_proc; //speed up searching
    struct spinlock lock;
    int period;
};

extern struct RBTREE *rdy_queue;

```

接下来用这些函数完成红黑树的初始化，插入与删除操作，头文件里的声明如下所示：

```

//cfs.h
void rotateLeft(struct RBTREE *tree, struct proc *ptr);
void rotateRight(struct RBTREE *tree, struct proc *ptr);
int insertProc(struct RBTREE *tree, struct proc *node);
struct proc* rdyqueue_deq(struct RBTREE *tree);
int preemptionTest(struct proc* current, struct proc* min_vruntime);
extern const int nice_weights[40];
struct proc* min_vruntime_proc(struct proc* node);
int getWeight(int nice);
void rbinit(struct RBTREE *tree);

```

讨论红黑树的实现是没有教益的，这是ADS课程的内容。最值得关心的函数是preemptionTest，它用于判断进程是否可以被抢占，简化后的代码如下所示：

```

int preemptionTest(struct proc* current, struct proc* min_vruntime) {
    if(time_used >= min_granularity) {
        if ((time_used >= current->timeSlice)) {
            return 1;
        }
        if (min_vruntime != 0 && current->vruntime > min_vruntime->vruntime)
        {
            if ((time_used != 0) && (time_used >= min_granularity)) {
                return 1;
            } else if (time_used == 0) {
                return 1;
            }
        }
    }
    return 0;
}

```

显然释放CPU只有在运行了最小时间粒度之后才有可能。事实上经典的CFS调度算法并不能提供实时系统的time bound，只能保障公平性，因此我们要求每个进程一旦运行就至少持续一个最小时间单位以避免starvation，剩下的结论在之前理论解释里已经陈述过了。

最后的内容就是考察进程入队和出队的时机。显然在任何一个可能会把进程设置为runnable，例如wakeup, fork, clone的时候，把进程插入rdy_queue是理智之选。

```
//proc.c:fork
    acquire(&wait_lock);
    np->parent = p;
    release(&wait_lock);
    acquire(&np->lock);
    np->state = RUNNABLE;
    insertProc(rdy_queue, np);
    release(&np->lock);
```

最关键的问题是进程何时离开就绪队列，在开始着手实现CFS，我们最迷惑的问题是就绪队列里的进程除了在被调度的时候之外，有没有可能变成不是runnable的？

例如kill会不会把一个就绪的进程杀死，让它在就绪队列里就变成zombie？如果这样的话，原有的红黑树删除的逻辑就不能直接适用我们的问题，甚至取出进程时也不一是要取树的最左结点，这从根本上破坏了就绪队列的性质。

好在这种考虑目前是多疑的，kill并不能直接杀死一个就绪的进程，如果可以的话，假若这个进程正在修改一个重要的内核数据结构，那么直接杀死它毫无疑问是危险的。我们只需要在被调度的时候把进程从优先队列里取出来即可。

以下就是简化版的调度器模型，调度器会从就绪队列中取出队头(也就是红黑树的最左结点)，如果有就让它运行，简化后的代码如下所示：

```
void
scheduler(void)
{
    for(;;){
        p = rdyqueue_deq(rdy_queue);
        if(p != 0){
            if(p->state == RUNNABLE) {
                p->state = RUNNING;
                swtch(&c->context, &p->context);
            }
        }
    }
}
```

当进程被时钟中断后，则需要判断是否要释放CPU，逻辑如下列代码所示：

```
void
yield(void)
{
    p->vruntime = p->vruntime + ((p->curruntime << 10) / p->weight);
    p->actual_time += p->curruntime;
```

```

    p->curruntime = 0;
    if(preemptionTest(proc, rdy_queue->min_vruntime_proc) == 1){
        p->state = RUNNABLE;
        insertProc(rdy_queue, p);
        sched();
    }
}

```

4 测试

我们准备了两个测试，一个简单但快捷，另一个耗时较长。

4.1 testCFS测试

如下代码是第一份测试，进入YuanShen OS后执行命令testCFS即可运行。测试的核心思想是让不同优先级的进程完成同一份任务，观察完成顺序，实际运行时间与虚拟运行时间。

在进行这份测试的时候，我们花费大量的时间找到并解决了红黑树删除时的一个BUG，即在删除根结点的时候有一处没有判断根结点是否为NullPointer。

测试代码如下所示:

```

#define NUM_PROCESSES 10

void child_process(int id, int priority) {
    set_priority(priority); // 设置优先级

    for (volatile int j = 0; j < 1000000000; j++)
        for(volatile int k = 0; k < 5; k++);

    // 打印进程信息和计数
    printf("Process %d (Priority %d)\n", id, priority);

    // 获取并打印运行时间和虚拟运行时间
    int runtime = get_runtime();
    int vruntime = get_vruntime();
    printf("get priority: %d\n", get_priority());
    printf("Process %d: Runtime = %d μs, VRuntime = %d μs\n", id, runtime,
vruntime);

    // 模拟一些工作

    exit(0); // 结束子进程
}

```

执行后可以得到如下结果，再次执行时可能结果略有偏差，属于随机因素的影响，总体结果应当大致一致:

```

$ testCFS
Process 1 (Priority 0)
get priority: 0
Process 1: Runtime = 1540000 µs, VRuntime = 1540000 µs
Process 10 (Priority 9)
get priority: 9
Process 10: Runtime = 1550000 µs, VRuntime = 11585320 µs
Process 9 (Priority 8)
get priority: 8
Process 9: Runtime = 1560000 µs, VRuntime = 9287304 µs
Process 8 (Priority 7)
get priority: 7
Process 8: Runtime = 1560000 µs, VRuntime = 7429812 µs
Process 7 (Priority 6)
get priority: 6
Process 7: Runtime = 1560000 µs, VRuntime = 5872932 µs
Process 6 (Priority 5)
get priority: 5
Process 6: Runtime = 1560000 µs, VRuntime = 4768452 µs
Process 5 (Priority 4)
get priority: 4
Process 5: Runtime = 1560000 µs, VRuntime = 3776448 µs
Process 4 (Priority 3)
get priority: 3
Process 4: Runtime = 1560000 µs, VRuntime = 3036852 µs
Process 2 (Priority 1)
get priority: 1
Process 2: Runtime = 1560000 µs, VRuntime = 1947972 µs
Process 3 (Priority 2)
get priority: 2
Process 3: Runtime = 1610000 µs, VRuntime = 2516913 µs
All child processes completed.

```

Fig2. testCFS测试运行结果

可以观察到对于同一份任务，不同的进程消耗的时间是相近的。vruntime和runtime的关系也符合我们的预期。而的确nice值越高的任务获得最少的时间片与调度权重(priority值越大反而优先级越低，不如把它理解为迟缓和度，或者理解为cpu-bound任务)。

4.2 testCFS测试 (intensive version)

然后启用第二份测试, 同样也位于testCFS，不过要先将之前的宏SIMPLE_TEST注释掉:

```

/*user/testCFS.c*/

#define SIMPLE_TEST

```

第二份测试会运行50个进程，每十个一组共享一个nice，分别为从0到4。输出的数据格式为(nice, runtime, vruntime)，运行结果如下图所示：

```
init: starting sh
$ testCFS
0, 1510000, 1510000
4, 1530000, 3703824
4, 1540000, 3728032
4, 1550000, 3752240
4, 1530000, 3703824
4, 1530000, 3703824
4, 1550000, 3752240
4, 1560000, 3776448
4, 1610000, 3897488
4, 1630000, 3945904
4, 1590000, 3849072
3, 1580000, 3075786
3, 1550000, 3017385
3, 1560000, 3036852
3, 1550000, 3017385
3, 1560000, 3036852
3, 1560000, 3036852
3, 1560000, 3036852
3, 1570000, 3056319
3, 1570000, 3056319
3, 1610000, 3134187
3, 1590000, 3095253
2, 1570000, 2454381
2, 1570000, 2454381
2, 1570000, 2454381
2, 1590000, 2485647
2, 1560000, 2438748
2, 1570000, 2454381
2, 1570000, 2454381
2, 1580000, 2470014
2, 1600000, 2516913
2, 1610000, 2516913
1, 1640000, 2047868
1, 1640000, 2047868
1, 1610000, 2010407
1, 1580000, 1972946
1, 1630000, 2035381
1, 1610000, 2010407
1, 1630000, 2032894
1, 1560000, 1947972
1, 1580000, 1972946
1, 1570000, 1960459
0, 1550000, 1550000
0, 1520000, 1520000
0, 1520000, 1520000
0, 1520000, 1520000
0, 1530000, 1530000
0, 1520000, 1520000
0, 1520000, 1520000
0, 1510000, 1510000
0, 1510000, 1510000
All child processes completed.
```

Fig3. testCFS加强版测试的运行结果

可以观察到nice值较低的任务被优先调度(有一个进程提早运行结束是因为它最先占据了时间片), vruntime和runtime的关系也符合理论计算值。

参考资料

[1] "进程调度分析（一）：CFS调度器原理及实现." <https://zhuanlan.zhihu.com/p/713838653>.

[2] Chiu CC. "红黑树." Second Round, 2016-01-23. <https://alrightchiu.github.io/SecondRound/red-black-tree-introjian-jie.html>.

[3] Rendell, Alistair. "Red Black Tree."

https://users.cecs.anu.edu.au/~Alistair.Rendell/Teaching/apac_comp3600/module3/red_black_trees.xhtml.