Linux 进程管理

目录

1. 基础知识

进程类型

- 实时进程
- 非实时进程

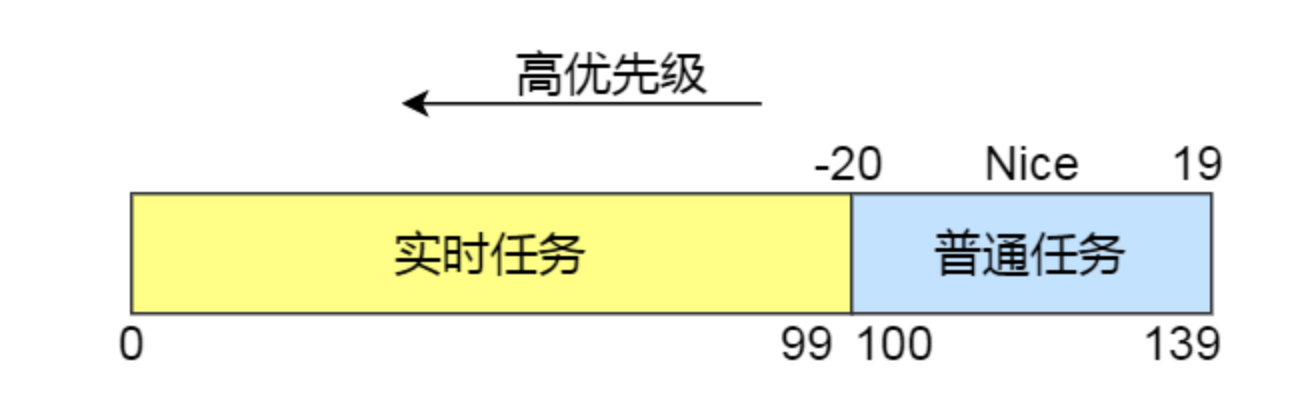
CPU 资源占用类型

- CPU 消耗性 （CPU-Bound）
- I/O 消耗性 （IO-Bound)
    - 交互进程（对响应时间有要求）

进程优先级

静态优先级 (static_prio)

- 实时进程 0 - 99
- 非实时进程 100 - 139



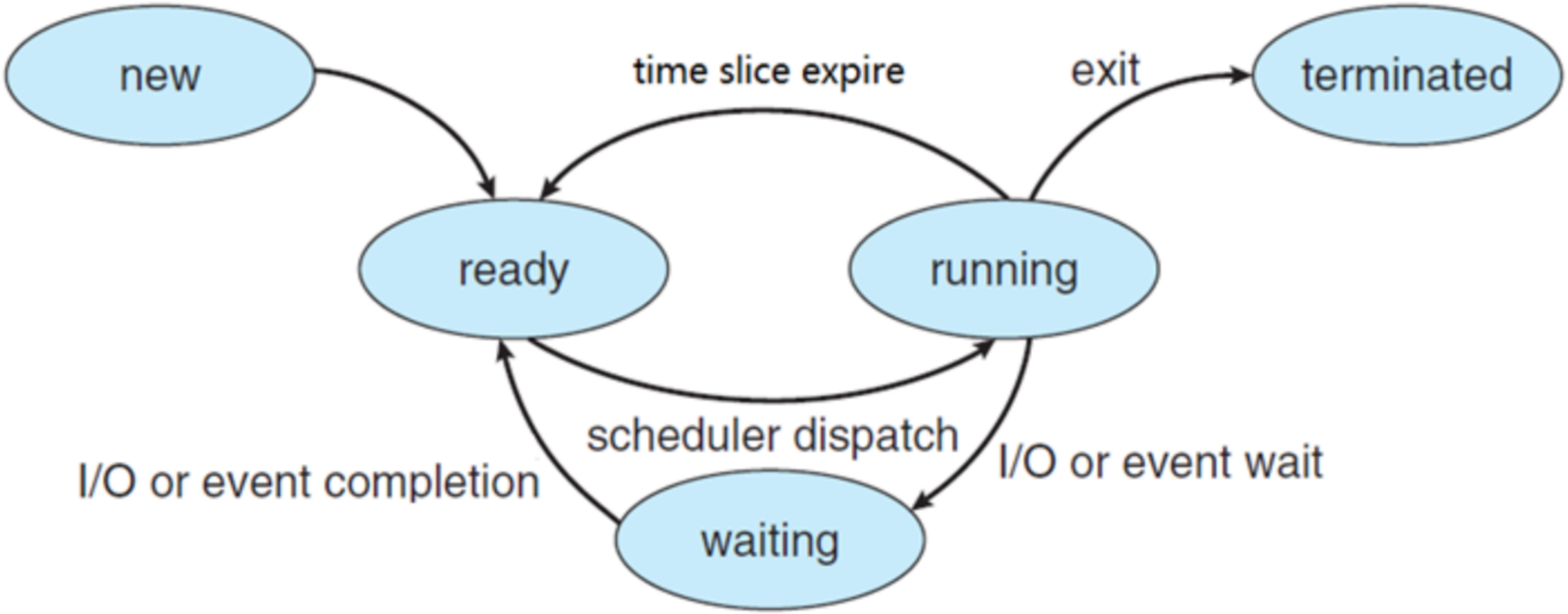
非实时进程静态优先级可以通过 nice() 进行调整。 范围 （-20 - 19）

动态优先级 (prio)
保存进程的动态优先级，通常也是调度类使用的优先级，某些情况下在内核中会临时调整该优先级，比如实时互斥量。

时间片 （time slice)
表示进程被调度进来和被调度出去所能持续的时间长度；

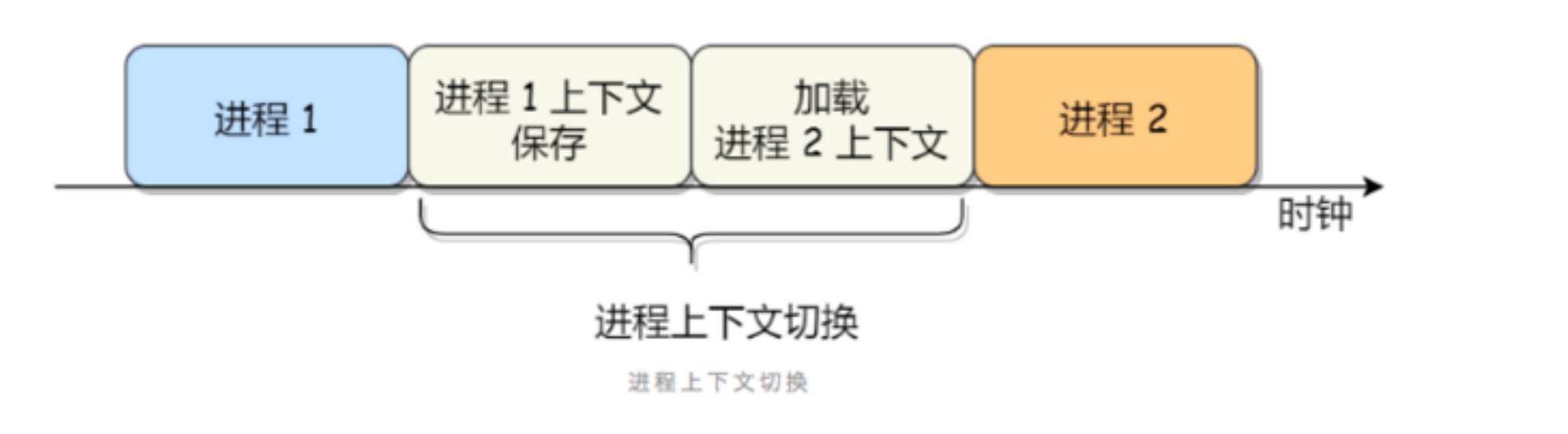
进程内核结构
进程在内核中数据结构 PCB（Process Control Block） 为 task_struct 结构。

进程运行状态


new
time slice expire
exit
terminated
ready
running
scheduler dispatch
I/O or event completion
I/O or event wait
waiting

进程上下文切换

进程的上下文切换不仅包含了虚拟内存、栈、全局变量等用户空间的资源，还包括了内核堆栈、寄存器等内核空间的资源。



参见 [操作系统进程知识总结]

2. Linux 调度器

2.1 调度器

调度器是整个系统的发动机，负责所有就绪进程的调度运行，设计良好的调度器需要从以下几个方面进行考虑：

- 吞吐量
- 响应时间
- 公平性
- 功耗 （移动设备）

调度类型

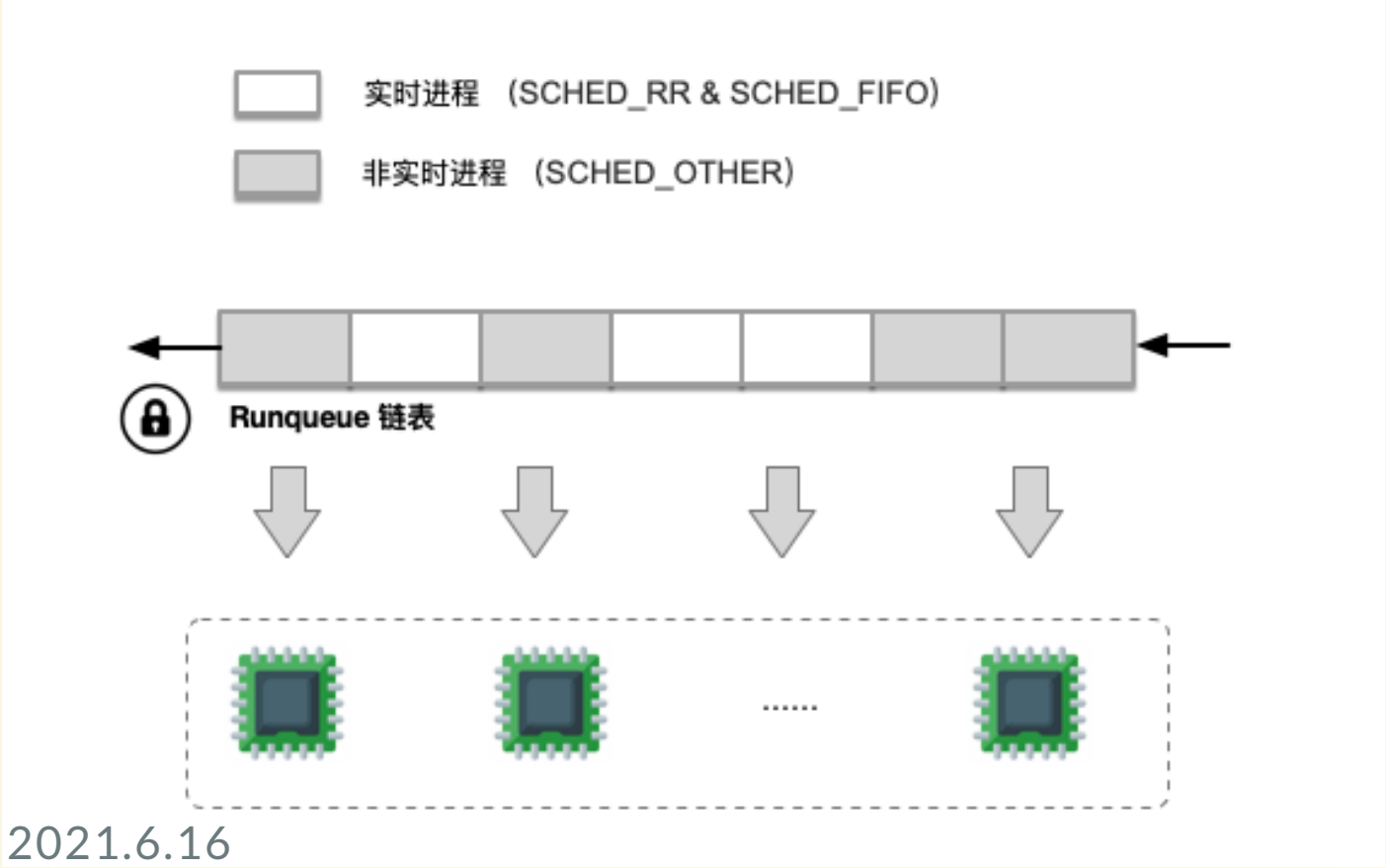
1. 主动放弃 CPU （由于需要等待或者主动进入 Sleep)
2. 被动放弃 CPU （由于时间片消耗完或被更高优先级的任务抢占-如支持抢占）

2.2 Linux 调度器历史

- O(n) 2.4 内核
- O(1) 2.6 内核
- CFS 2.6.23+ (当前默认调度器)

2.3 O(n) 调度算法

调度数据结构



2.3 O(n) 调度算法

调度数据结构

复杂度：时间复杂度为 O(n), n 代表当前就绪状态的进程数目。

全局一个列表队列，不区分实时和非实时进程。CPU 遍历的时候都需要加锁操作，当就绪任务较多时，加锁周期会比较长。

进程就绪状态加入到 runqueue 队列，阻塞或者退出时候从队列移除。

时间片分配

在运行时动态计算优先级，保证实时进程的优先级高于非实时任务的动态优先级，实时进程动态优先级 （1000 + p->rt_priority）; 在计算动态优先级的时候会综合考虑该任务的剩余时间片、历史执行情况等因素。

任务 Nice 越低，剩余时间片越多，动态优先级越高，越有机会获得运行；

任务运行的最大时间由动态优先级转变为固定 tick 数目，tick 的时间值由系统动态配置的 HZ 时钟精度确定。

	−20	−10	0	10	19
HZ=100	11个tick 110ms	8个tick 80ms	6个tick 60ms	3个tick 30ms	1个tick 10ms
HZ=1000	81个tick 81ms	61个tick 61ms	41个tick 41ms	21tick 21ms	3个tick 3ms

调度运行

调度器遍历整个就绪队列中全部的 n 个任务，选取 **动态优先级** 最高的任务进行调度运行。

```
        list_for_each(tmp, &runqueue_head) {
                p = list_entry(tmp, struct task_struct, run_list);
                if (can_schedule(p, this_cpu)) {
                        int weight = goodness(p, this_cpu, prev->active_mm);
                        if (weight > c)
                                c = weight, next = p;
                }
        }
```

goodness函数中找出优先级最高的一个进程。

调度运行

tick 中断：tick 中断函数中检查检查时间片是否使用完，使用完毕则进行进程切换；否则将剩余时间片的 1/2 追加到动态优先级。

当系统无可运行任务或所有任务执行完时间片后，调度器需要将**系统中**（不是 runqueue 中）所有任务的重新设置默认时间片；如果等待队列中的进程的时间片仍然有剩余，则会折半计入到下一次的时间片中；

优点

实现逻辑简单，在早期运行任务数量有限的情况下，运行良好。

缺点

- 时间复杂度为 O(n)，当所有调度任务完成后，需要对全部任务重新初始化，耗时高；
- 实时进程可能不能及时调度；
- 全局只有一个运行队列，会导致 CPU 存在空闲；
- SMP 系统中扩展问题，访问运行队列需要加锁，特别是进程数量大时；
- 进程在不同 CPU 上进行跳转，cache 缺失，性能受到影响；

Linux内核中不确定性的 N 可能会带来较大的风险影响。

2.4 O(1) 调度器

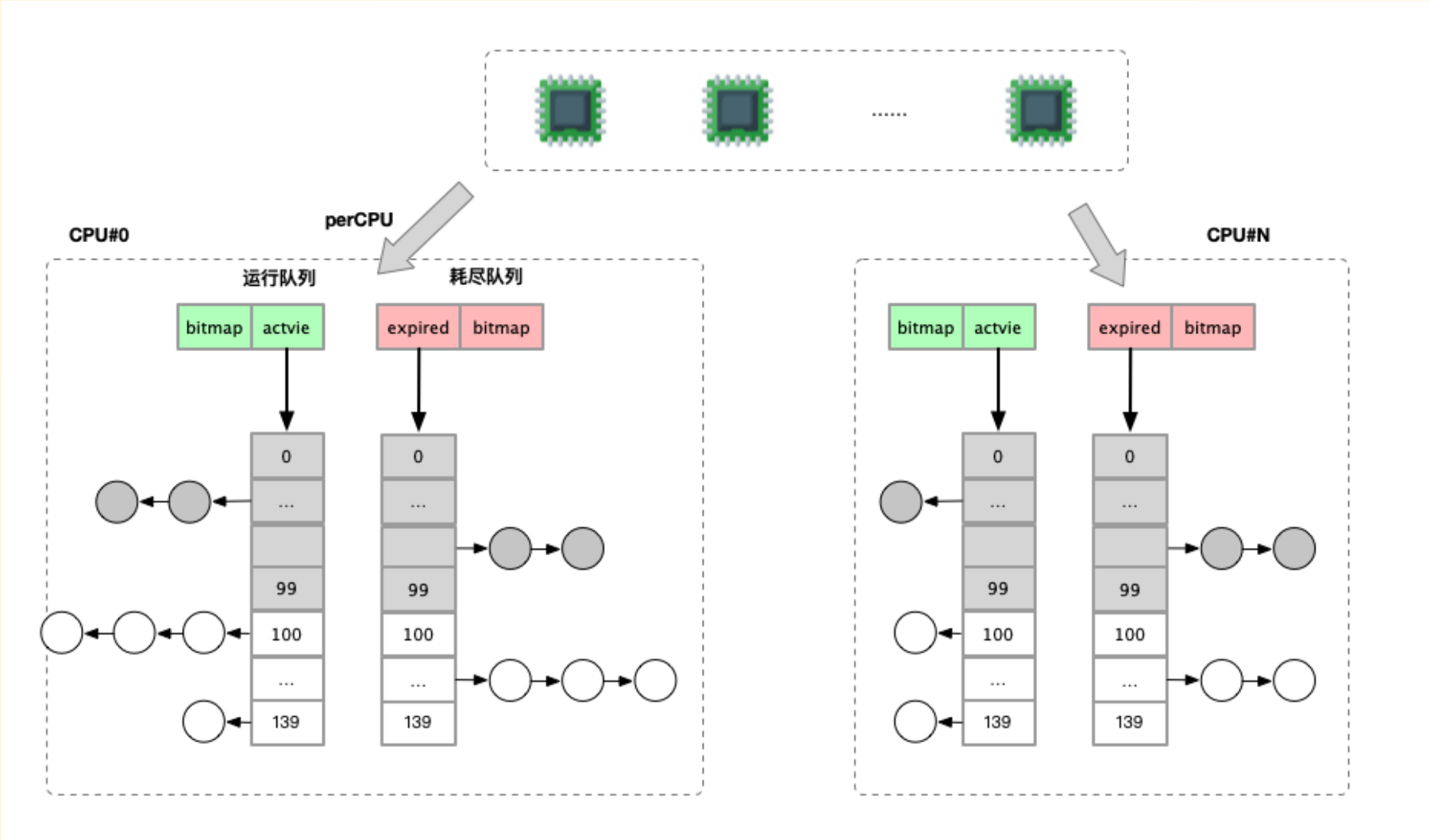
数据结构

每个 CPU 一个优先级队列的方式。同时对于队列设置一个按照进程优先级的数组。

分为 active 和 expired 两个数组，分别表示当前正常运行的队列和已经过期的队列。

当 active 队列任务全部完成运行后，将 expired 与 active 互换。

为了快速判断优先级数组的列表中是否存在任务，使用 bitmap 索引。


.....
perCPU
CPU#0
CPU#N
运行队列
耗尽队列
bitmap
actvie
expired
bitmap
0
...
99
100
...
139

时间片分配

静态优先级为 [-20 ... 0 ... 19] 的普通进程，其缺省时间片为[800ms ... 100ms ... 5ms]。

同时结合进程的实际运行情况，比如在 runqueue 中的等待时间、睡眠时间等因素，实时调整动态优先级，以期望更好识别出交互进程。

runqueue 中的进程优先级为动态优先级。

动态优先级映射为时间片。

调度运行

schedule 函数的主要功能是从该 CPU 的 runqueue 找到一个最合适的进程调度执行。其基本的思路就是从 active 优先级队列中依次查找，代码如下：

```
idx = sched_find_first_bit(array->bitmap);
queue = array->queue + idx;
next = list_entry(queue->next, task_t, run_list);
```

当 active 优先级队列中没有需要运行的任务，则交换 active 与 expired。

不再需要像 O(n) 那样对系统中全部任务重新初始化时间片。

在 tick 函数中断中判断任务时间片是否使用完，如果时间片使用完毕，正常情况下将任务从移动到 expired 队列中，但是如果判断为交互进程，依照交互进程指数 TASK_INTERACTIVE ，可能还是会放入到 actvie 队列中；

> 如果 expired 队列中的任务等待时间过长，说明调度其严重不公，这时候即使判断出来时间片耗尽的进程是交互进程，仍然会被放入 expired 队列，尽快完成本次调度；

优点

O(1) 调度器开销比较小，大多数场景下都有不错的表现；

重点关注交互进程，在交互进程的响应时间方面得到了很好的改善；

缺点

- 交互式任务的判定算法过于复杂，参数硬编码，代码可维护性变差；
- 静态时间片，任务增加调度延时上升；

2.5 CFS 调度器

调度策略

进程调度依赖于调度策略，Linux 把相同调度策略抽象成了调度类（schedule class)。CFS 只负责非实时进程的处理。

优先级按照从高到低的方式进行调度， STOP 最高， IDLE 最低。

进程权重

CFS 调度器中使用静态优先级对应的权重概念来对应。权重的对应关系（-20 - 19）。进程每降低一个 nice 值，将多获得 10% CPU 的时间。

```
const int sched_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,
/*  15 */        36,        29,        23,        18,        15,
};
```

vruntime（virtual runtime)

表示该进程已经在 CPU 上运行的时间，值越大被再次调度概率越低。
将不同优先级的进程运行的时间作为一个统一视图来进程处理。

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

- `vruntime` 虚拟运行时间
- `delta_exec` 真实运行时间
- `nice_0_weight` 表示 `nice` 为 0 的进程权重
- `weight` 表示进程权重

- 进程的 vruntime 增长速度取决于其优先级
  - 优先级越低，则 vruntime 增长越快，因而被再次调用的可能性就越小
  - 优先级越高，则 vruntime 增长越慢，因而被再次调用的可能性就越大

动态时间片

调度周期 sched_period，保证一个调度周期内，运行队列中的所有任务都会被调度一次，最坏情况下，任务的调度延时为一个调度周期；

默认调度周期 sched_latency 默认为 6 ms；任务平均时间片最小值 sched_min_granularity 为 0.75 ms

8 个以内可运行时间设定为 6ms，否则为 N * sched_min_granularity；

```
if sched_latency/nr_running < sched_min_granularity
     sched_period = nr_running * sched_min_granularity
```

确定好 sched_period 以后，每个任务的动态时间片为：

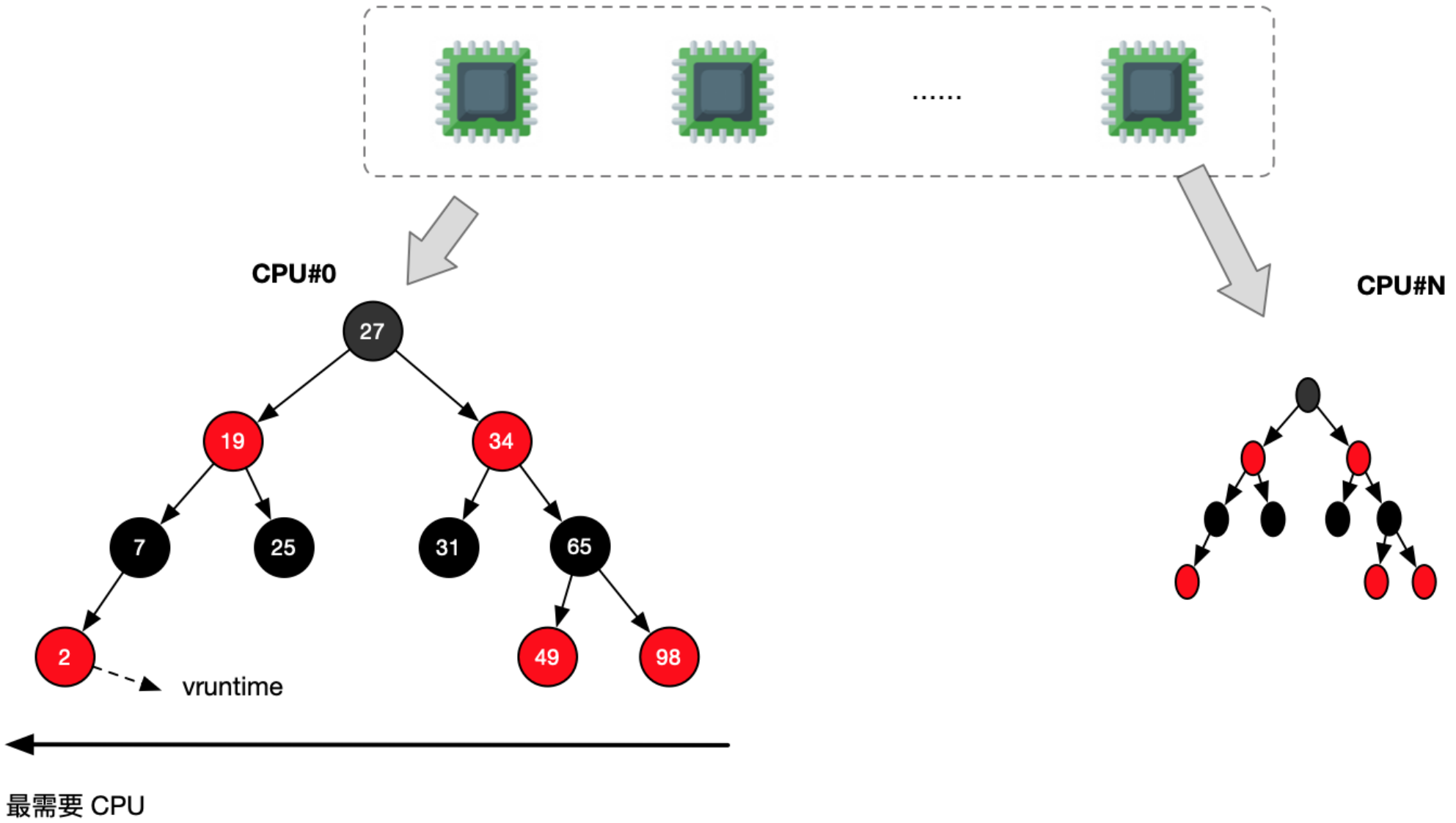
$$time\_slice\_i = \frac{sched\_period * weight}{sum\_weight\_rq}$$

调度数据结构

每个 CPU 对应一个运行队列 runqueue，该运行队列中包含 dl_rq/rt_rq/cfs_rq 等多个调度队列。CFS 只处理普通进程。

使用 vruntime 作为红黑树的索引键，使用 sched_entity 作为值。

最小值在最左侧，可以 O(1) 进行遍历，插入 O(logN)


.....
CPU#0
CPU#N
27
19
34
7
25
31
65
2
49
98
vruntime
最需要 CPU

调度运行

```
static inline struct task_struct *
pick_next_task(struct rq *rq, struct task_struct *prev, struct rq_flags *rf)
{
        struct task_struct *p;
        if (likely((prev->sched_class == &idle_sched_class ||
                    prev->sched_class == &fair_sched_class) &&
                   rq->nr_running == rq->cfs.h_nr_running)) {

                p = fair_sched_class.pick_next_task(rq, prev, rf);
        }
again:
        for_each_class(class) {
                p = class->pick_next_task(rq, prev, rf);
        }
}
```

pick_next_task_fair

```
static struct task_struct *
pick_next_task_fair(struct rq *rq, struct task_struct *prev, struct rq_flags *rf)
{
        struct cfs_rq *cfs_rq = &rq->cfs;
        struct sched_entity *se;
        struct task_struct *p;
        put_prev_task(rq, prev);

        do {
                se = pick_next_entity(cfs_rq, NULL);
                set_next_entity(cfs_rq, se);
                cfs_rq = group_cfs_rq(se);
        } while (cfs_rq);

        p = task_of(se);
        return p;
}
```

pick_next_entity 从红黑树上读取调度实体

```
static struct sched_entity *
pick_next_entity(struct cfs_rq *cfs_rq, struct sched_entity *curr)
{
        struct sched_entity *left = __pick_first_entity(cfs_rq);
        struct sched_entity *se;

        se = left; /* ideally we run the leftmost entity */

    // .. 抢占/唤醒或者其他判断

        return se;
}
```

调度函数在各自 CPU 上执行，依次运行 dl_rq/rt_rq/cfs_rq 队列。

运行 cfs_rq 队列时：

- 每次调度程序都选择红黑树最左侧的节点运行(其 virtual runtime 最小)，将其从红黑树中删除，并调整红黑树
- 每隔一段时间，就更新正在运行的进程的 vruntime，并将更新后的值与当前红黑树中最左侧节点的 vruntime 相比较
  - 如果小于最左侧节点的 virtual runtime，则继续执行当前进程
  - 如果大于最左侧节点的 virtual runtime，则抢占式地运行最左侧的节点，并将当前正在运行的进程重新放到红黑树中

而且在每一个调度周期内调度器会不断检查公平性是否得到了满足，通过修订 vruntme 的值来进行平衡，比如新加入的进程和被唤醒的进程会在 vruntme 上进行补偿。

新进程加入及睡眠进程唤醒

```
static void place_entity(struct cfs_rq *cfs_rq, struct sched_entity *se, int initial)
{
        u64 vruntime = cfs_rq->min_vruntime;

        if (initial && sched_feat(START_DEBIT)) // 如果是初始化
                vruntime += sched_vslice(cfs_rq, se); // 计算新进程的 vruntime 添加到 min_vruntime，防止过度占用

        if (!initial) {
                unsigned long thresh = sysctl_sched_latency; // 一个调度周期延时

                if (sched_feat(GENTLE_FAIR_SLEEPERS))
                        thresh >>= 1;  // 折半处理，防止过度占用

                vruntime -= thresh;  // 减掉 thresh，让进程获得尽快调度
        }

        /* ensure we never gain time by being placed backwards. */
        se->vruntime = max_vruntime(se->vruntime, vruntime);
}
```

优点

基于优先级，兼顾运行公平性，默认调度器，经受住了时间的考验；

缺点

在 Android 等嵌入式设备上，仍然需要调整和优化；

调度器综合对比

项/调度算法	O(N)	O(1)	CFS
数据结构	全局一个队列	per-CPU List bitmap + active + expired	per-CPU 红黑树
是否包含实时进程	是	是	否
进程时间片	静态权重映射为 tick 数目	静态权重映射为固定 ms	静态优先级 => 权重 => 时间片ms
调度依据	动态优先级	动态优先级	最小 vruntime
调度周期	就绪进程 tick 总和	就绪进程 ms 总和	sched_latency/nr_running * sched_min_granularity
调度运行	动态优先级遍历	动态优先级遍历	最小 vruntime 调度，保证公平
tick 时间中断	检查时间片	检查时间片	检查时间片和公平性
睡眠进程	剩余的 CPU 会折半累加到下次调度周期	依据平均睡眠时间提升动态优先级	调度周期补偿 vruntme - 1/2*sched_latency
调度周期结束	全部进程参与初始化	无	无

未来调度器

ARM 体系存在 big.LITTLE 模型，存在大小核数的结构。

- HMP 调度器
- EAS 绿色节省调度器 （HMP 的优化版）
- NUMA 调度器

3. 高阶话题 - 抢占

1. 触发抢占： 给正在CPU上运行的当前进程设置一个请求重新调度的标志(TIF_NEED_RESCHED)，仅此而已，此时进程并没有切换。

触发时机：

- 周期性的时钟中断： scheduler_tick() 时间片耗尽
- 进程唤醒：当进程被唤醒的时候，优先级高于CPU当前进程
- 新进程创建
- 进程修改 nice 值
- 负载均衡

2. 执行抢占：在随后的某个时刻，内核会检查TIF_NEED_RESCHED标志并调用 __schedule()执行抢占。

抢占如果发生在进程处于用户态的时候，称为 User Preemption（用户态抢占）；如果发生在进程处于内核态的时候，则称为Kernel Preemption（内核态抢占）。

- 从系统调用(syscall)返回用户态时
- 中断处理程序返回内核空间之前会检查TIF_NEED_RESCHED标志，如果置位则调用preempt_schedule_irq()执行抢占。preempt_schedule_irq()是对schedule()的包装。

抢占只在某些特定的时机发生，这是内核的代码决定的。

3. 高阶话题 - CPU 负载计算方式 PLET

为了让调度器更加高效，需要针对每个调度队列和每个调度实体的负载进行衡量。（Per Entity Load Tracking）

时间（物理时间，不是虚拟时间）被分成了 1024us 的序列，在每一个周期中，一个 Entity 对系统负载的贡献可以根据该实体处于 Runnable 状态（正在 CPU 上运行或者等待 CPU 调度运行）的时间进行计算。如果在该周期内，Runnable的时间是 x，那么对系统负载的贡献就是（x/1024）。

负载需要考虑历史周期的影响 y^32 = 0.5, y = 0.97857206

```
L = L0 + L1 * y + L2 * y^2 + L3 * y^3 + ... + Ln * y^n
```

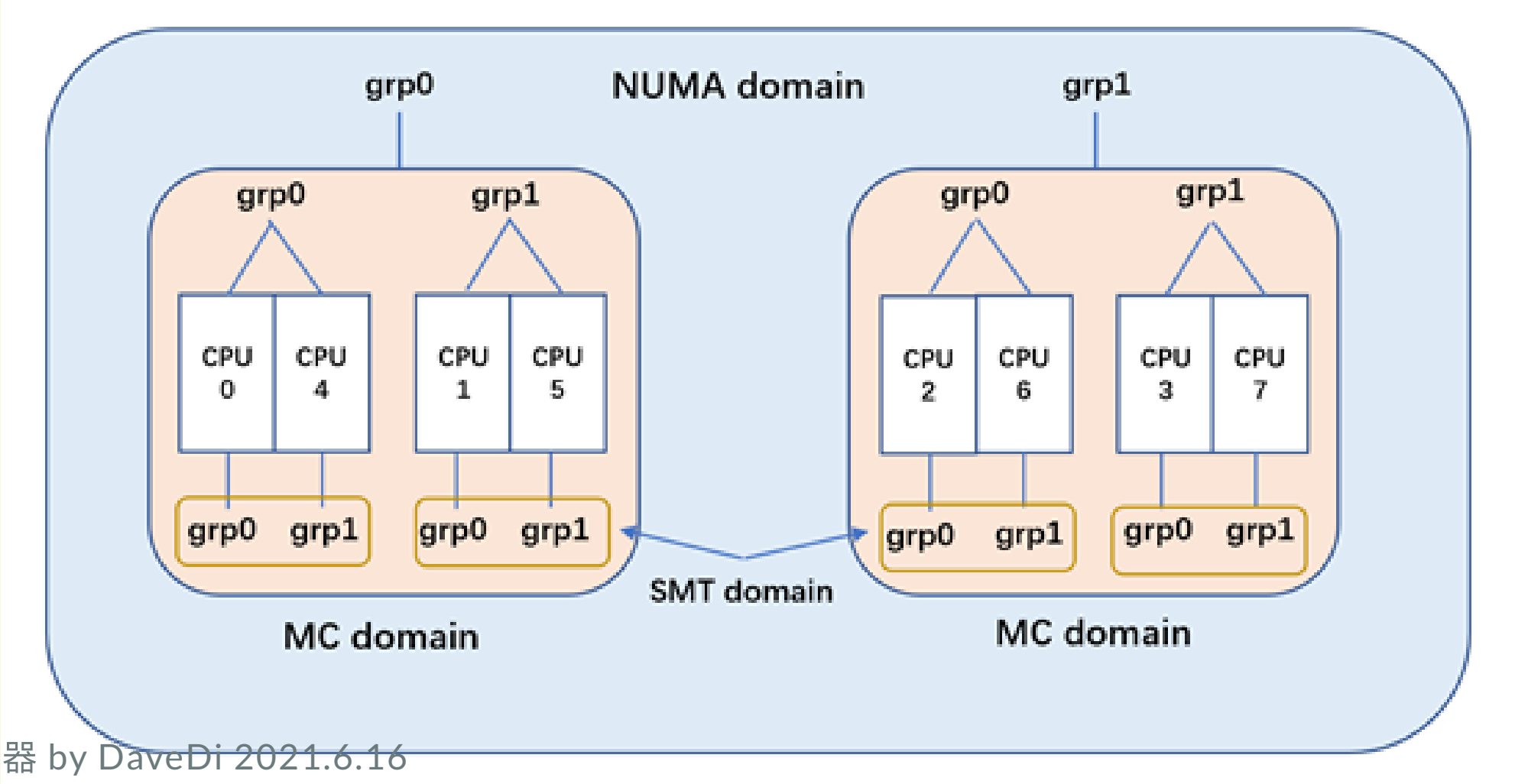
当前对于负载的计算 上个周期的负载贡献值乘以衰减系数，加上当前时间点的负载即可。

> 经过 32 个周期，负载降低为 0.5 倍, 经过 2016 个周期，全部负载衰减为 0

更新时机：就绪队列添加任务、删除任务和周期性更新

就绪队列的 load_avg（包含睡眠进程的贡献） 与 runnable_load_avg（只统计可运行任务的负载）。

3. 高阶话题 - 负载均衡 SMP 和 NUMA



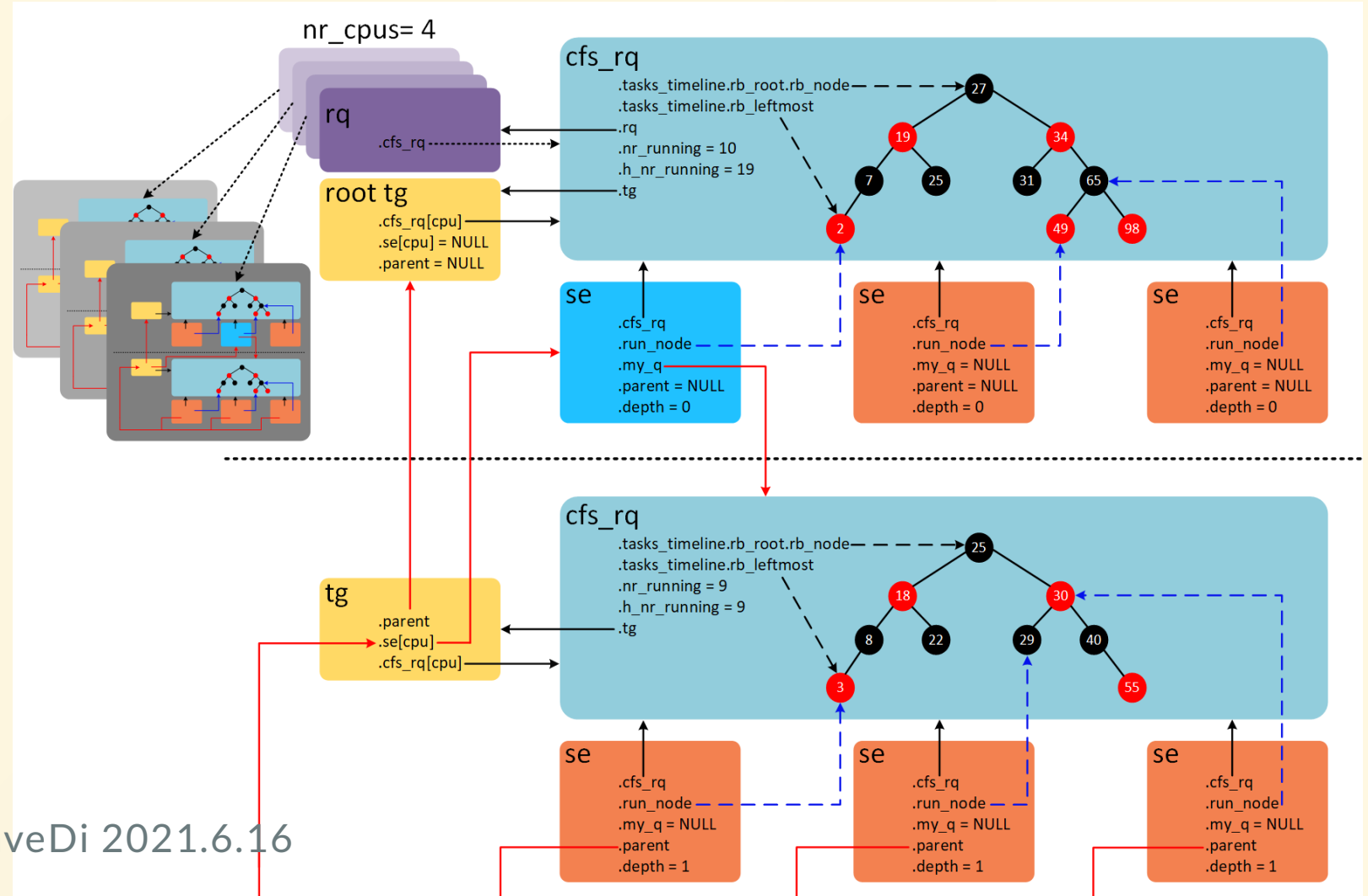
负载均衡迁移基于 runnable_load_avg 的值。

负载均衡流程：

1. 判断当前 CPU 是否需要负载均衡
2. 负载均衡以当前 CPU 开始，由下至上调度域，从最底层的调度域开始做负载均衡
3. 在调度域中查找最繁忙的调度组，更新调度域和调度组的相信统计信息，计算出该调度域的不均衡负载值；
4. 在最繁忙的调度组中找出最繁忙的 CPU，然后把繁忙的 CPU 中的任务迁移到当前的 CPU，迁移负载量为不均衡负载值。

NUMA 类似，更是更加复杂，需要处理跨 NUMA 内存和进程迁移。

3. 高阶话题 - 调度任务组


nr_cpus= 4
rq
.cfs_rq
root tg
.cfs_rq[cpu]
.se[cpu] = NULL
.parent = NULL
cfs_rq
.tasks_timeline.rb_root.rb_node
.tasks_timeline.rb_leftmost
.rq
.nr_running = 10
.h_nr_running = 19
.tg
27
19
34
7
25
31
65
2
49
98
se
.cfs_rq
.run_node
.my_q
.parent = NULL
.depth = 0
se
.cfs_rq
.run_node
.my_q = NULL
.parent = NULL
.depth = 0
se
.cfs_rq
.run_node
.my_q = NULL
.parent = NULL
.depth = 0
tg
.parent
.se[cpu]
.cfs_rq[cpu]
cfs_rq
.tasks_timeline.rb_root.rb_node
.tasks_timeline.rb_leftmost
.nr_running = 9
.h_nr_running = 9
.tg
25
18
30
8
22
29
40
3
55
se
.cfs_rq
.run_node
.my_q = NULL
.parent
.depth = 1
se
.cfs_rq
.run_node
.my_q = NULL
.parent
.depth = 1
se
.cfs_rq
.run_node
.my_q = NULL
.parent
.depth = 1

4. Q & A

Thank You!