负载计算

schedule里面这个负载(load average)的概念常常被理解成cpu占用率，这个有比较大的偏差。schedule不使用cpu占用率来评估负载，而是使用平均时间runnable的数量来评估负载。Shedule也分了几个层级来计算负载：

- entity级负载计算：update_load_avg()
- cpu级负载计算：update_cpu_load_active()
- 系统级负载计算：calc_global_load_tick()

计算负载的目的是为了去做负载均衡，下面我们逐个介绍各个层级负载算法和各种负载均衡算法。

一、PELT(Per-Entity Load Tracking)Entity级的负载计算

1. Entity级的负载计算也称作PELT(Per-Entity Load Tracking)。
2. 注意负载计算时使用的时间都是实际运行时间而不是虚拟运行时间vruntime。

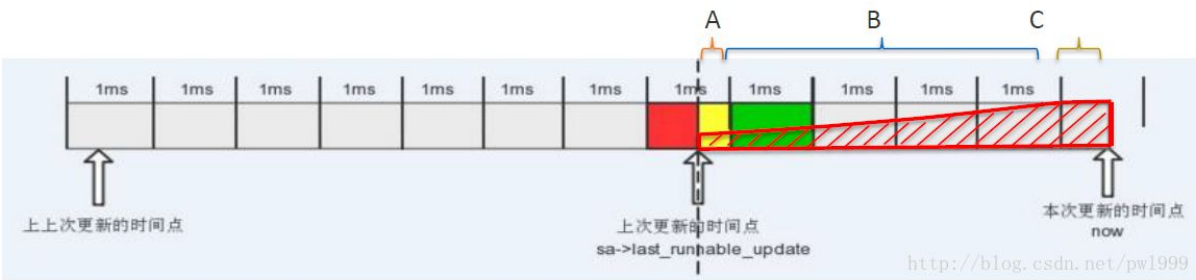
```
scheduler_tick() -> task_tick_fair() -> entity_tick() -> update_load_avg():

↓

/* Update task and its cfs_rq load average */
static inline void update_load_avg(struct sched_entity *se, int update_tg)
{
    struct cfs_rq *cfs_rq = cfs_rq_of(se);
    u64 now = cfs_rq_clock_task(cfs_rq);
    int cpu = cpu_of(rq_of(cfs_rq));
    unsigned long runnable_delta = 0;
    unsigned long prev_load;
    int on_rq_task = entity_is_task(se) && se->on_rq;

    if (on_rq_task) {
#ifdef CONFIG_MTK_SCHED_RQAVG_US
        inc_nr_heavy_running("__update_load_avg-", task_of(se), -1, false);
#endif
        prev_load = se_load(se);
    }
    /*
     * Track task load average for carrying it to new CPU after migrated, and
     * track group sched_entity load average for task_h_load calc in migration
     */
     /* (1) 计算se的负载 */
    __update_load_avg(now, cpu, &se->avg,
              se->on_rq * scale_load_down(se->load.weight),
              cfs_rq->curr == se, NULL);

#ifdef CONFIG_MTK_SCHED_RQAVG_US
    if (entity_is_task(se) && se->on_rq)
        inc_nr_heavy_running("__update_load_avg+", task_of(se), 1, false);
#endif

    /* (2) 计算cfs_rq的负载 */
    if (update_cfs_rq_load_avg(now, cfs_rq) && update_tg)
        update_tg_load_avg(cfs_rq, 0);
```

```
    /* sched: add trace_sched */
    if (entity_is_task(se)) {
        trace_sched_task_entity_avg(1, task_of(se), &se->avg);
        trace_sched_load_avg_task(task_of(se), &se->avg);
    }

    if (on_rq_task) {
        runnable_delta = prev_load - se_load(se);
#ifdef CONFIG_HMP_TRACER
        trace_sched_cfs_load_update(task_of(se), se_load(se), runnable_delta, cpu);
#endif
    }

    trace_sched_load_avg_cpu(cpu, cfs_rq);
}

|→

/* Group cfs_rq's load_avg is used for task_h_load and update_cfs_share */
static inline int update_cfs_rq_load_avg(u64 now, struct cfs_rq *cfs_rq)
{

    /* (2.1) 同样调用__update_load_avg()函数来计算cfs_rq的负载 */
    decayed = __update_load_avg(now, cpu_of(rq_of(cfs_rq)), sa,
        scale_load_down(cfs_rq->load.weight), cfs_rq->curr != NULL, cfs_rq);

}
```

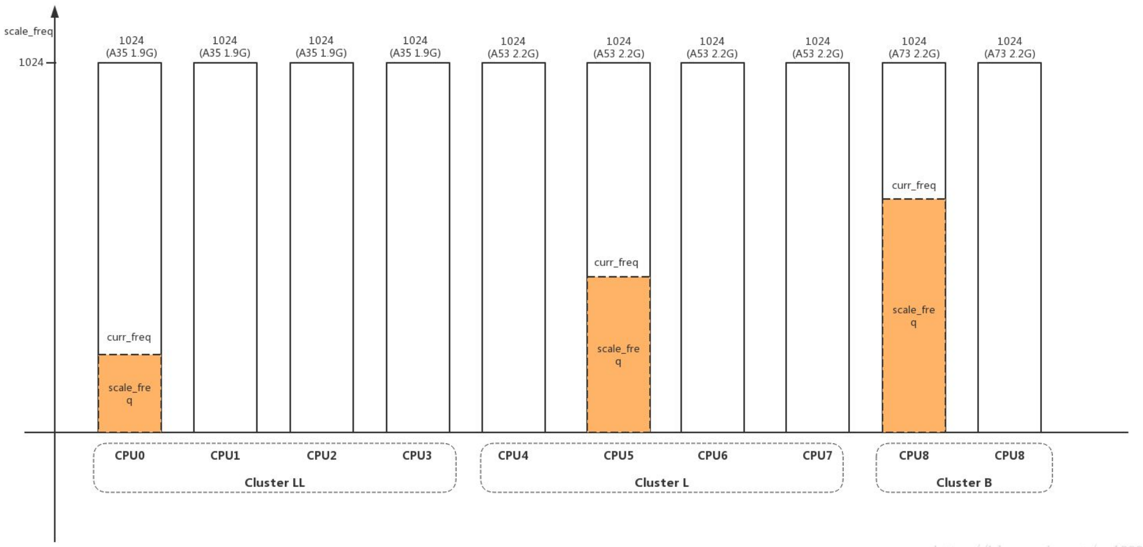
二、核心函数__update_load_avg()

__update_load_avg()函数是计算负载的核心，核心思想还是求一个相对值。这时1024变量又登场了，前面说过因为内核不能表示分数，所以把1扩展成1024。和负载相关的各个变量因子都使用1024来表达相对能力：时间、weight(nice优先级)、cpufreq、cpucapacity。

- 1、等比队列(geometric series)的求和；把时间分成1024us(1ms)的等分。除了当前等分，过去等分负载都要进行衰减，linux引入了衰减比例 y = 0.978520621，y^32 = 0.5。也就是说一个负载经过1024us(1ms)以后不能以原来的值参与计算了，要衰减到原来的0.978520621倍，衰减32个1024us(1ms)周期以后达到原来的0.5倍。每个等分的衰减比例都是不一样的，所以最后的负载计算变成了一个等比队列(geometric series)的求和。等比队列的特性和求和公式如下(y即是公式中的等比比例q)：
- 时间分段；在计算一段超过1024us(1ms)的时间负载时，__update_load_avg()会把需要计算的时间分成3份：时间段A和之前计算的负载补齐1024us，时间段B是多个1024us的取整，时间段C是最后不能取整1024us的余数。



- scale_freq、scale_cpu的含义；scale_freq表示 当前freq 相对 本cpu最大freq 的比值：scale_freq = (cpu_curr_freq / cpu_max_freq) * 1024：



通项公式

$$a_n = a_1 \times q^{(n-1)}$$

求和公式推导

（1） $S_n=a_1+a_2+a_3+...+a_n$(公比为q)

（2） $qS_n=a_1q + a_2q + a_3q +...+ a_nq = a_2+ a_3+ a_4+...+ a_n+ a_{(n+1)}$

（3） $S_n-qS_n=(1-q)S_n=a_1-a_{(n+1)}$

（4） $a_{(n+1)}=a_1q^n$

（5） $S_n=a_1(1-q^n)/(1-q)$ （q≠1）[1]

求和公式

$$S_n = na_1, (q=1)$$

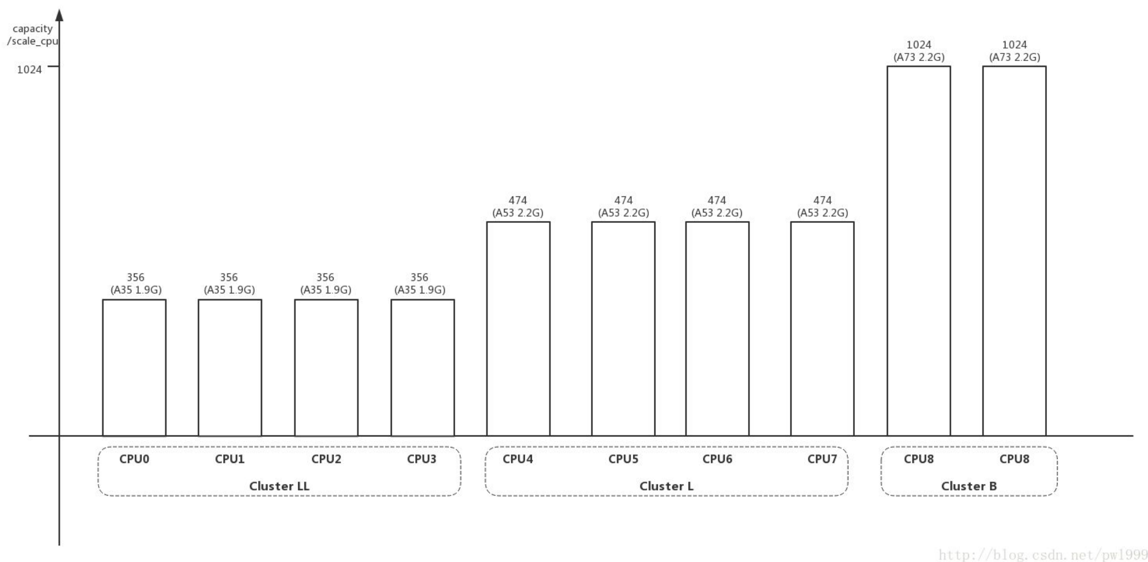
$$S_n = \frac{a_1 \times (1-q^n)}{1-q} = \frac{a_1 - a_n q}{1-q} = \frac{a_n q - a_1}{q-1}, (q \neq 1)$$

$$S_\infty = \frac{a_1}{1-q} (|q| < 1, n \to \infty)$$


```
static __always_inline int
__update_load_avg(u64 now, int cpu, struct sched_avg *sa,
          unsigned long weight, int running, struct cfs_rq *cfs_rq)
{

    scale_freq = arch_scale_freq_capacity(NULL, cpu);

}
```

```
↓

unsigned long arch_scale_freq_capacity(struct sched_domain *sd, int cpu)
{
    unsigned long curr = atomic_long_read(&per_cpu(cpu_freq_capacity, cpu));

    if (!curr)
        return SCHED_CAPACITY_SCALE;

    /* (1) 返回per_cpu(cpu_freq_capacity, cpu) */
    return curr;
}

void arch_scale_set_curr_freq(int cpu, unsigned long freq)
{
    unsigned long max = atomic_long_read(&per_cpu(cpu_max_freq, cpu));
    unsigned long curr;

    if (!max)
        return;

    /* (1.1) cpu的 cpu_curr_freq / cpu_max_freq * 1024 */
    curr = (freq * SCHED_CAPACITY_SCALE) / max;

    atomic_long_set(&per_cpu(cpu_freq_capacity, cpu), curr);
}
```

scale_cpu表示 (当前cpu最大运算能力 相对 所有cpu中最大的运算能力 的比值) * (cpufreq_policy的最大频率 相对 本cpu最大频率 的比值)，：scale_cpu = cpu_scale * max_freq_scale / 1024。后续的rebalance计算中经常使用capacity的叫法，和scale_cpu是同一含义。因为max_freq_scale基本=1024，所以scale_cpu基本就是cpu_scale的值：




```
unsigned long arch_scale_cpu_capacity(struct sched_domain *sd, int cpu)
{
#ifdef CONFIG_CPU_FREQ
    unsigned long max_freq_scale = cpufreq_scale_max_freq_capacity(cpu);
```

```
    return per_cpu(cpu_scale, cpu) * max_freq_scale >> SCHED_CAPACITY_SHIFT;
#else
    return per_cpu(cpu_scale, cpu);
#endif
}
```

cpu_scale表示 当前cpu最大运算能力 相对 所有cpu中最大的运算能力 的比值：cpu_scale = ((cpu_max_freq * efficiency) / max_cpu_perf) * 1024

当前cpu的最大运算能力等于当前cpu的最大频率乘以当前cpu每clk的运算能力efficiency, efficiency相当于DMIPS，A53/A73不同架构每个clk的运算能力是不一样的：

```
/* (1.1) 不同架构的efficiency */
static const struct cpu_efficiency table_efficiency[] = {
    { "arm,cortex-a73", 3630 },
    { "arm,cortex-a72", 4186 },
    { "arm,cortex-a57", 3891 },
    { "arm,cortex-a53", 2048 },
    { "arm,cortex-a35", 1661 },
    { NULL, },
};

static void __init parse_dt_cpu_capacity(void)
{

    for_each_possible_cpu(cpu) {

        rate = of_get_property(cn, "clock-frequency", &len);

        /* (1) 计算当前cpu的perf能力 = clkrate * efficiency */
        cpu_perf = ((be32_to_cpup(rate)) >> 20) * cpu_eff->efficiency;
        cpu_capacity(cpu) = cpu_perf;

        /* (2) 计算soc中最强cpu的perf能力max_cpu_perf
         */
        max_cpu_perf = max(max_cpu_perf, cpu_perf);
        min_cpu_perf = min(min_cpu_perf, cpu_perf);

    }
}

static void update_cpu_capacity(unsigned int cpu)
{
    unsigned long capacity = cpu_capacity(cpu);

#ifdef CONFIG_MTK_SCHED_EAS_PLUS
    if (cpu_core_energy(cpu)) {
#else
    if (0) {
#endif
        /* if power table is found, get capacity of CPU from it */
        int max_cap_idx = cpu_core_energy(cpu)->nr_cap_states - 1;

        /* (3.1) 使用查表法得到相对perf能力cpu_scale */
```

```
            capacity = cpu_core_energy(cpu)->cap_states[max_cap_idx].cap;
        } else {
            if (!capacity || !max_cpu_perf) {
                cpu_capacity(cpu) = 0;
                return;
            }

            /* (3.1) 使用计算法得到相对perf能力cpu_scale,
                cpu_scale = (capacity / max_cpu_perf) * 1024
             */
            capacity *= SCHED_CAPACITY_SCALE;
            capacity /= max_cpu_perf;
        }
        set_capacity_scale(cpu, capacity);
    }

static void set_capacity_scale(unsigned int cpu, unsigned long capacity)
{
    per_cpu(cpu_scale, cpu) = capacity;
}
```

例如mt6799一共有10个cpu，为“4 A35 + 4 A53 + 2 A73”架构。使用计算法计算的cpu_scale相关值：

```
/* rate是从dts读取的和实际不符合，只是表达一下算法 */
cpu = 0, rate = 1190, efficiency = 1661, cpu_perf = 1976590
cpu = 1, rate = 1190, efficiency = 1661, cpu_perf = 1976590
cpu = 2, rate = 1190, efficiency = 1661, cpu_perf = 1976590
cpu = 3, rate = 1190, efficiency = 1661, cpu_perf = 1976590
cpu = 4, rate = 1314, efficiency = 2048, cpu_perf = 2691072
cpu = 5, rate = 1314, efficiency = 2048, cpu_perf = 2691072
cpu = 6, rate = 1314, efficiency = 2048, cpu_perf = 2691072
cpu = 7, rate = 1314, efficiency = 2048, cpu_perf = 2691072
cpu = 8, rate = 1562, efficiency = 3630, cpu_perf = 5670060
cpu = 9, rate = 1562, efficiency = 3630, cpu_perf = 5670060
```

mt6799实际是使用查表法直接得到cpu_scale的值：

```
struct upower_tbl upower_tbl_ll_1_FY = {
    .row = {
        {.cap = 100, .volt = 75000, .dyn_pwr = 9994, .lkg_pwr = {13681, 13681, 13681, 13681, 13681, 13681} },
        {.cap = 126, .volt = 75000, .dyn_pwr = 12585, .lkg_pwr = {13681, 13681, 13681, 13681, 13681, 13681} },
        {.cap = 148, .volt = 75000, .dyn_pwr = 14806, .lkg_pwr = {13681, 13681, 13681, 13681, 13681, 13681} },
        {.cap = 167, .volt = 75000, .dyn_pwr = 16656, .lkg_pwr = {13681, 13681, 13681, 13681, 13681, 13681} },
        {.cap = 189, .volt = 75000, .dyn_pwr = 18877, .lkg_pwr = {13681, 13681, 13681, 13681, 13681, 13681} },
        {.cap = 212, .volt = 75000, .dyn_pwr = 21098, .lkg_pwr = {13681, 13681, 13681, 13681, 13681, 13681} },
        {.cap = 230, .volt = 75700, .dyn_pwr = 23379, .lkg_pwr = {13936, 13936, 13936, 13936, 13936, 13936} },
        {.cap = 245, .volt = 78100, .dyn_pwr = 26490, .lkg_pwr = {14811, 14811, 14811, 14811, 14811, 14811} },
        {.cap = 263, .volt = 81100, .dyn_pwr = 30729, .lkg_pwr = {15958, 15958, 15958, 15958, 15958, 15958} },
```

```
        {.cap = 278, .volt = 83500, .dyn_pwr = 34409, .lkg_pwr = {16949,
16949, 16949, 16949, 16949, 16949} },
        {.cap = 293, .volt = 86000, .dyn_pwr = 38447, .lkg_pwr = {18036,
18036, 18036, 18036, 18036, 18036} },
        {.cap = 304, .volt = 88400, .dyn_pwr = 42166, .lkg_pwr = {19159,
19159, 19159, 19159, 19159, 19159} },
        {.cap = 319, .volt = 90800, .dyn_pwr = 46657, .lkg_pwr = {20333,
20333, 20333, 20333, 20333, 20333} },
        {.cap = 334, .volt = 93200, .dyn_pwr = 51442, .lkg_pwr = {21605,
21605, 21605, 21605, 21605, 21605} },
        {.cap = 345, .volt = 95000, .dyn_pwr = 55230, .lkg_pwr = {22560,
22560, 22560, 22560, 22560, 22560} },
        {.cap = 356, .volt = 97400, .dyn_pwr = 59928, .lkg_pwr = {24002,
24002, 24002, 24002, 24002, 24002} },
    },
    .lkg_idx = DEFAULT_LKG_IDX,
    .row_num = UPOWER_OPP_NUM,
    .nr_idle_states = NR_UPOWER_CSTATES,
    .idle_states = {
        {{0}, {7321} },
        {{0}, {7321} },
        {{0}, {7321} },
        {{0}, {7321} },
        {{0}, {7321} },
        {{0}, {7321} },
    },
};
```

max_freq_scale表示 cpufreq_policy的最大频率 相对 本cpu最大频率 的比值：

max_freq_scale = (policy->max / cpuinfo->max_freq) * 1024：

```
static void
scale_freq_capacity(struct cpufreq_policy *policy, struct cpufreq_freqs
*freqs)
{

    scale = (policy->max << SCHED_CAPACITY_SHIFT) / cpuinfo->max_freq;

    for_each_cpu(cpu, &cls_cpus)
        per_cpu(max_freq_scale, cpu) = scale;

}
```

- decay_load()；decay_load(val,n)的意思就是负载值val经过n个衰减周期(1024us)以后的值，主要用来计算时间段A即之前的值的衰减值。

```
/*
 * Approximate:
 *   val * y^n,    where y^32 ~= 0.5 (~1 scheduling period)
 */
static __always_inline u64 decay_load(u64 val, u64 n)
{
    unsigned int local_n;

    if (!n)
        return val;
```

```
    else if (unlikely(n > LOAD_AVG_PERIOD * 63))
        return 0;

    /* after bounds checking we can collapse to 32-bit */
    local_n = n;

    /* (1) 如果n是32的整数倍，因为2^32 = 1/2，相当于右移一位，
        计算n有多少个32，每个32右移一位
     */
    /*
     * As y^PERIOD = 1/2, we can combine
     *    y^n = 1/2^(n/PERIOD) * y^(n%PERIOD)
     * With a look-up table which covers y^n (n<PERIOD)
     *
     * To achieve constant time decay_load.
     */
    if (unlikely(local_n >= LOAD_AVG_PERIOD)) {
        val >>= local_n / LOAD_AVG_PERIOD;
        local_n %= LOAD_AVG_PERIOD;
    }

    /*  (2) 剩下的值计算 val * y^n，
        把y^n计算转换成 (val * runnable_avg_yN_inv[n] >> 32)
     */
    val = mul_u64_u32_shr(val, runnable_avg_yN_inv[local_n], 32);
    return val;
}

/* Precomputed fixed inverse multiplies for multiplication by y^n */
static const u32 runnable_avg_yN_inv[] = {
    0xffffffff, 0xfa83b2da, 0xf5257d14, 0xefe4b99a, 0xeac0c6e6, 0xe5b906e6,
    0xe0ccdeeb, 0xdbfbb796, 0xd744fcc9, 0xd2a81d91, 0xce248c14, 0xc9b9bd85,
    0xc5672a10, 0xc12c4cc9, 0xbd08a39e, 0xb8fbaf46, 0xb504f333, 0xb123f581,
    0xad583ee9, 0xa9a15ab4, 0xa5fed6a9, 0xa2704302, 0x9ef5325f, 0x9b8d39b9,
    0x9837f050, 0x94f4efa8, 0x91c3d373, 0x8ea4398a, 0x8b95c1e3, 0x88980e80,
    0x85aac367, 0x82cd8698,
};
```

- __compute_runnable_contrib()：decay_load()只是计算y^n，而__compute_runnable_contrib()是计算一个对比队列的和：y + y^2 + y^3 … + y^n。计算时间段B的负载。runnable_avg_yN_sum[]数组是使用查表法来计算n=32位内的等比队列求和:
  runnable_avg_yN_sum[1] = y^1 * 1024 = 0.978520621 * 1024 = 1002
  runnable_avg_yN_sum[1] = (y^1 + y^2) * 1024 = 1982
  …
  runnable_avg_yN_sum[1] = (y^1 + y^2 .. + y^32) * 1024 = 23371

```
/*
 * For updates fully spanning n periods, the contribution to runnable
 * average will be: \Sum 1024*y^n
 *
 * We can compute this reasonably efficiently by combining:
 *   y^PERIOD = 1/2 with precomputed \Sum 1024*y^n {for  n <PERIOD}
 */
static u32 __compute_runnable_contrib(u64 n)
```

```
{
    u32 contrib = 0;

    if (likely(n <= LOAD_AVG_PERIOD))
        return runnable_avg_yN_sum[n];
    else if (unlikely(n >= LOAD_AVG_MAX_N))
        return LOAD_AVG_MAX;

    /* (1) 如果n>32，计算32的整数部分 */
    /* Compute \Sum k^n combining precomputed values for k^i, \Sum k^j */
    do {
        /* (1.1) 每整数32的衰减就是0.5 */
        contrib /= 2; /* y^LOAD_AVG_PERIOD = 1/2 */
        contrib += runnable_avg_yN_sum[LOAD_AVG_PERIOD];

        n -= LOAD_AVG_PERIOD;
    } while (n > LOAD_AVG_PERIOD);

    /* (2.1) 将整数部分对余数n进行衰减 */
    contrib = decay_load(contrib, n);

    /* (2.2) 剩余余数n，使用查表法计算 */
    return contrib + runnable_avg_yN_sum[n];
}

/*
 * Precomputed \Sum y^k { 1<=k<=n }.  These are floor(true_value) to prevent
 * over-estimates when re-combining.
 */
static const u32 runnable_avg_yN_sum[] = {
        0, 1002, 1982, 2941, 3880, 4798, 5697, 6576, 7437, 8279, 9103,
     9909,10698,11470,12226,12966,13690,14398,15091,15769,16433,17082,
    17718,18340,18949,19545,20128,20698,21256,21802,22336,22859,23371,
};
```

- se->on_rq；在系统从睡眠状态被唤醒，睡眠时间会不会被统计进load_avg？答案是不会。系统使用了一个技巧来处理这种情况，调用__update_load_avg()函数时，第三个参数weight = se->on_rq * scale_load_down(se->load.weight)。运行状态时se->on_rq=1，weight>0，老负载被老化，新负载被累加；在进程从睡眠状态被唤醒时，se->on_rq=0，weight=0，只有老负载被老化，睡眠时间不会被统计；

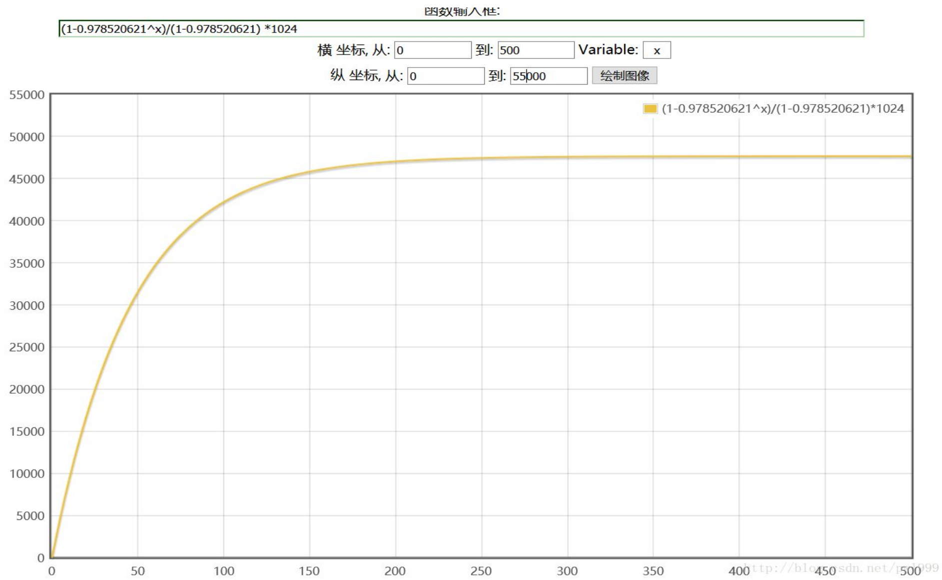
```
enqueue_task_fair()

|→

static void
enqueue_entity(struct cfs_rq *cfs_rq, struct sched_entity *se, int flags)
{

    /* (1) 在调用负载更新时，se->on_rq = 0 */
    enqueue_entity_load_avg(cfs_rq, se);

    se->on_rq = 1;

}
```

```
||→

static inline void
enqueue_entity_load_avg(struct cfs_rq *cfs_rq, struct sched_entity *se)
{

    if (!migrated) {

        /* (1.1) 传入weight=0, 只老化旧负载, 不统计新负载 */
        __update_load_avg(now, cpu_of(rq_of(cfs_rq)), sa,
            se->on_rq * scale_load_down(se->load.weight),
            cfs_rq->curr == se, NULL);
    }

}
```

相同的技巧是在更新cfs_rq负载时，调用__update_load_avg()函数时，第三个参数weight = scale_load_down(cfs_rq->load.weight)。如果cfs_rq没有任何进程时cfs_rq->load.weight=0，如果cfs_rq有进程时cfs_rq->load.weight=进程weight的累加值，这样在cfs没有进程idle时，就不会统计负载。但是如果被RT进程抢占，还是会统计(相当于cfs_rq的runnable状态)。

```
static inline int update_cfs_rq_load_avg(u64 now, struct cfs_rq *cfs_rq)
{
    struct sched_avg *sa = &cfs_rq->avg;
    int decayed, removed = 0;


    decayed = __update_load_avg(now, cpu_of(rq_of(cfs_rq)), sa,
        scale_load_down(cfs_rq->load.weight), cfs_rq->curr != NULL, cfs_rq);
}
```

- LOAD_AVG_MAX；从上面的计算过程解析可以看到，负载计算就是一个等比队列的求和。对于负载其实我们不关心他的绝对值，而是关心他和最大负载对比的相对值。所谓最大负载就是时间轴上一直都在，且能力值也都是最大的1(1024)。我们从上面等比队列的求和公式:Sn = a1(1-q^n)/(1-q) = 1024(1 - 0.978520621^n)/(1-0.978520621)。我们来看这个求和函数的曲线。



从曲线上分析，当x到达一定值后y趋于稳定，不再增长。利用这个原理linux定义出了负载最大值LOAD_AVG_MAX。含义是经过了LOAD_AVG_MAX_N(345)个周期以后，等比队列求和达到最大值LOAD_AVG_MAX(47742)：

```
/*
 * We choose a half-life close to 1 scheduling period.
 * Note: The tables runnable_avg_yN_inv and runnable_avg_yN_sum are
 * dependent on this value.
 */
#define LOAD_AVG_PERIOD 32
#define LOAD_AVG_MAX 47742 /* maximum possible load avg */
#define LOAD_AVG_MAX_N 345 /* number of full periods to produce LOAD_AVG_MAX
*/
```

平均负载都是负载和最大负载之间的比值：

```
static __always_inline int
__update_load_avg(u64 now, int cpu, struct sched_avg *sa,
          unsigned long weight, int running, struct cfs_rq *cfs_rq)
{


    if (decayed) {
        sa->load_avg = div_u64(sa->load_sum, LOAD_AVG_MAX);
        sa->loadwop_avg = div_u64(sa->loadwop_sum, LOAD_AVG_MAX);

        if (cfs_rq) {
            cfs_rq->runnable_load_avg =
                div_u64(cfs_rq->runnable_load_sum, LOAD_AVG_MAX);
            cfs_rq->avg.loadwop_avg =
                div_u64(cfs_rq->avg.loadwop_sum, LOAD_AVG_MAX);
        }
        sa->util_avg = sa->util_sum / LOAD_AVG_MAX;
    }
}
```

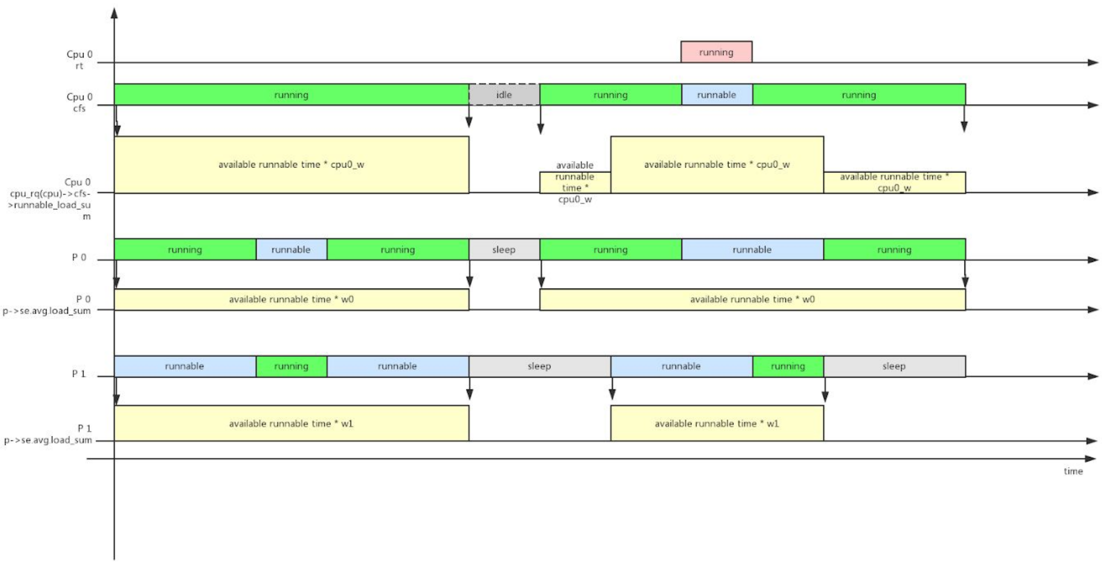
struct sched_avg数据成员的含义

```
/*
 * The load_avg/util_avg accumulates an infinite geometric series.
 * 1) load_avg factors frequency scaling into the amount of time that a
 * sched_entity is runnable on a rq into its weight. For cfs_rq, it is the
 * aggregated such weights of all runnable and blocked sched_entities.
 * 2) util_avg factors frequency and cpu scaling into the amount of time
 * that a sched_entity is running on a CPU, in the range
[0..SCHED_LOAD_SCALE].
 * For cfs_rq, it is the aggregated such times of all runnable and
 * blocked sched_entities.
 * The 64 bit load_sum can:
 * 1) for cfs_rq, afford 4353082796 (=2^64/47742/88761) entities with
 * the highest weight (=88761) always runnable, we should not overflow
 * 2) for entity, support any load.weight always runnable
 */
struct sched_avg {
    u64 last_update_time, load_sum;
    u32 util_sum, period_contrib;
    unsigned long load_avg, util_avg;
```

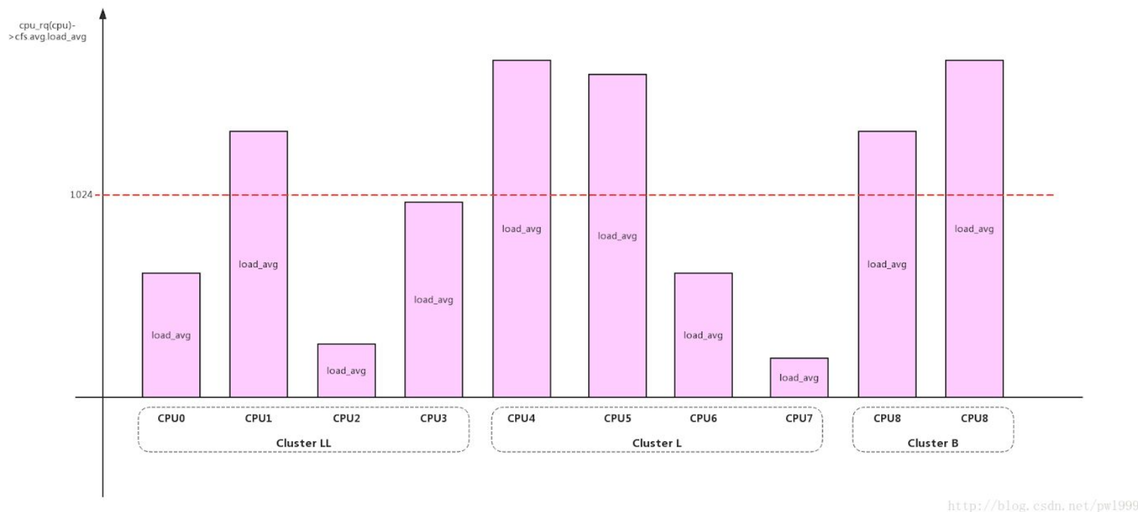
```
};
```

- load_avg:

1、runnable时间分量+weight分量的累加：avg.load_sum。
(load_sum累加了(runnable时间分量*scale_freq*weight)，不包含capacity分量)。

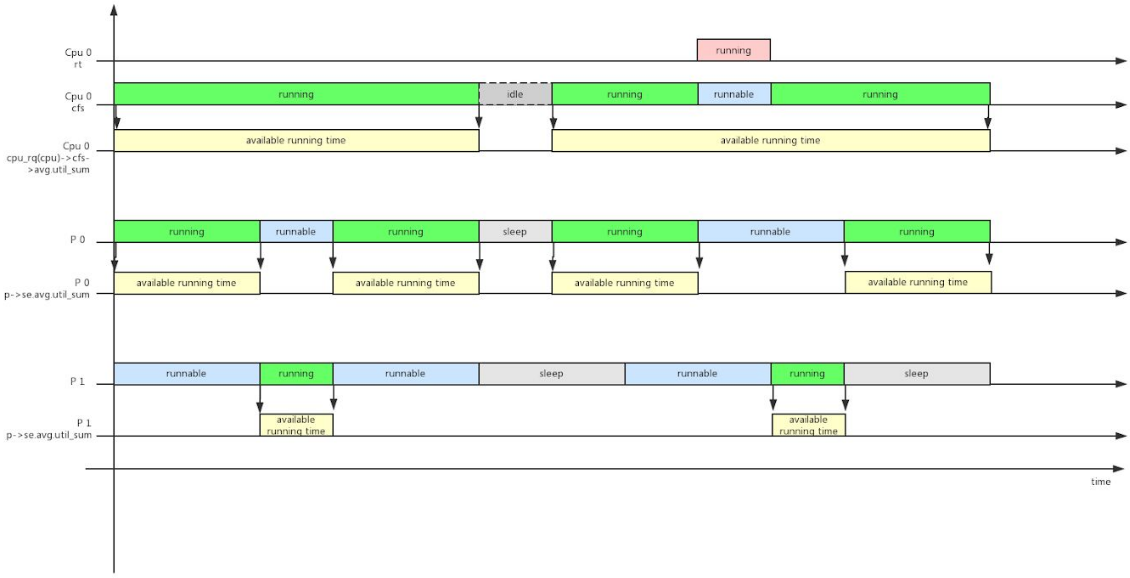


2、runnable时间分量+weight分量的平均值：avg.load_avg。
(avg->load_avg = div_u64(avg-->load_sum, LOAD_AVG_MAX);)。
因为weight分量的加入，load_avg突破1024的限制。但是遵循一个原则：cpu_rq(cpu)->cfs->avg.load_avg = p0->se.avg.load_sum + ... + pN->se.avg.load_sum

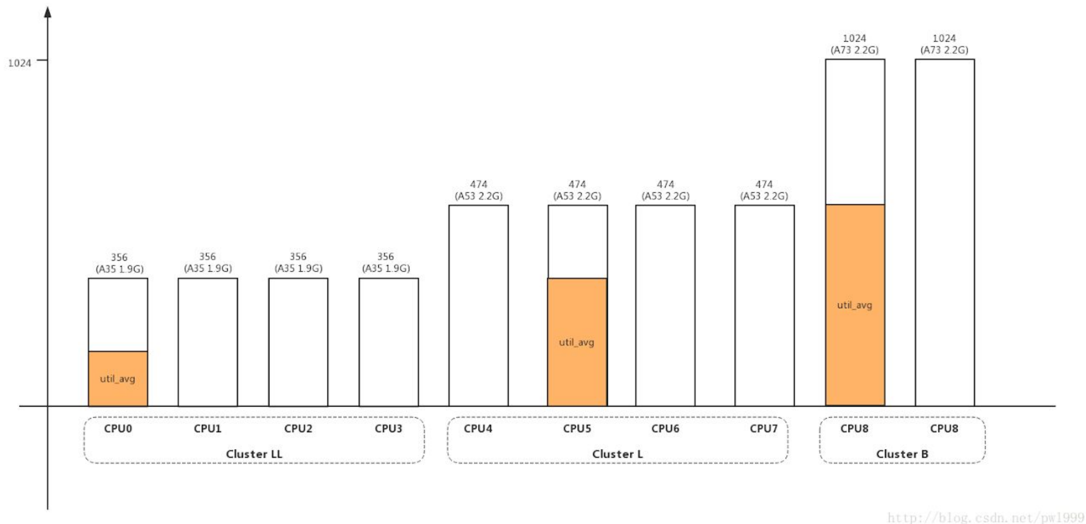


- util_avg

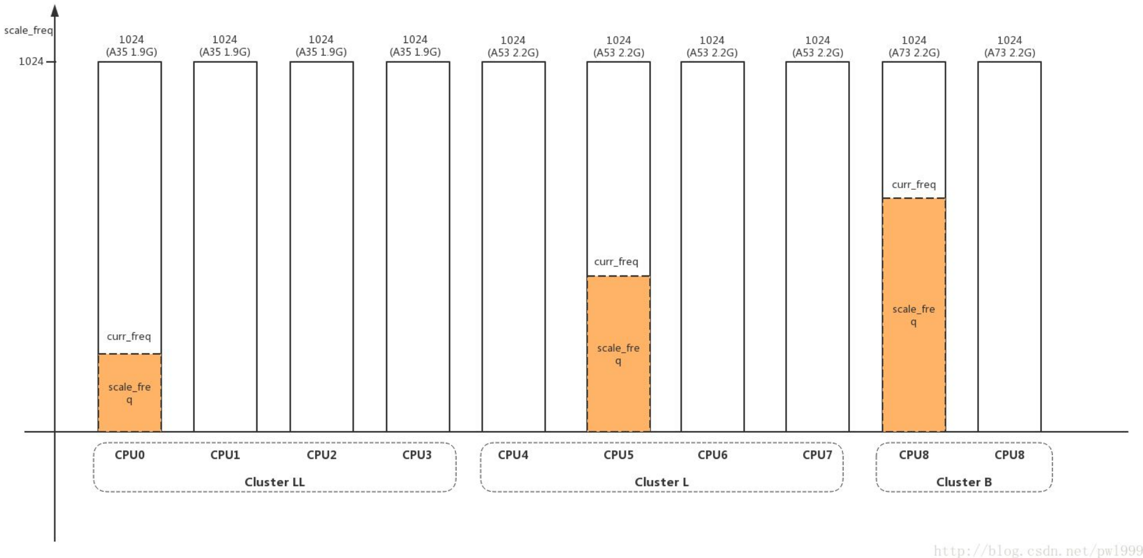
1、running时间分量的累加：avg.util_sum。
(loadwop_sum只累加(running时间分量*scale_freq*scale_cpu)，不包含weight分量、scale_cpu即capacity分量)。



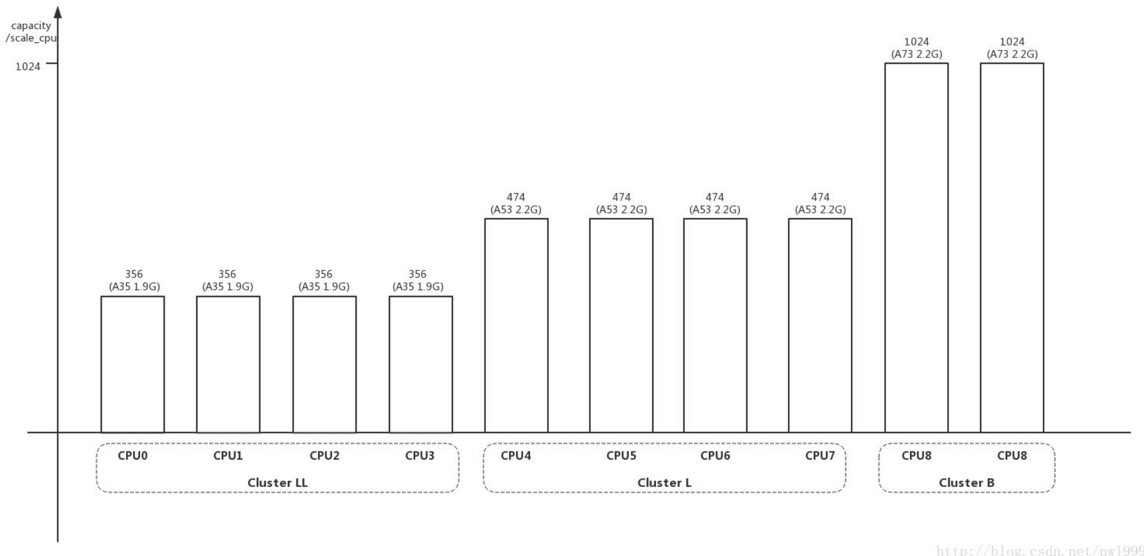
2、running时间分量的平均值：avg.util_avg。
(avg->util_avg = sa->util_sum / LOAD_AVG_MAX;)。



- Scale_freq，需要特别强调的是loadwop_avg、load_avg、util_avg在他们的时间分量中都乘以了scale_freq，所以上面几图都是他们在max_freq下的表现，实际的负载还受当前freq的影响：



- capacity/scale_cpu



- capacity是在smp负载均衡时更新：

```
run_rebalance_domains() -> rebalance_domains() -> load_balance() -> find_busiest_group() -> update_sd_lb_stats() -> update_group_capacity() -> update_cpu_capacity()

↓

static void update_cpu_capacity(struct sched_domain *sd, int cpu)
{
    unsigned long capacity = arch_scale_cpu_capacity(sd, cpu);
    struct sched_group *sdg = sd->groups;
    struct max_cpu_capacity *mcc;
    unsigned long max_capacity;
    int max_cap_cpu;
    unsigned long flags;

    /* (1) cpu_capacity_orig = cpu最大频率时的最大capacity */
```

```
    cpu_rq(cpu)->cpu_capacity_orig = capacity;

    mcc = &cpu_rq(cpu)->rd->max_cpu_capacity;

    raw_spin_lock_irqsave(&mcc->lock, flags);
    max_capacity = mcc->val;
    max_cap_cpu = mcc->cpu;

    if ((max_capacity > capacity && max_cap_cpu == cpu) ||
        (max_capacity < capacity)) {
        mcc->val = capacity;
        mcc->cpu = cpu;
#ifdef CONFIG_SCHED_DEBUG
        raw_spin_unlock_irqrestore(&mcc->lock, flags);
        /* pr_info("CPU%d: update max cpu_capacity %lu\n", cpu, capacity);
*/
        goto skip_unlock;
#endif
    }
    raw_spin_unlock_irqrestore(&mcc->lock, flags);

skip_unlock: __attribute__ ((unused));
    capacity *= scale_rt_capacity(cpu);
    capacity >>= SCHED_CAPACITY_SHIFT;

    if (!capacity)
        capacity = 1;

    /* (2) cpu_capacity = 最大capacity减去rt进程占用的比例 */
    cpu_rq(cpu)->cpu_capacity = capacity;
    sdg->sgc->capacity = capacity;
}
```

获取capacity的函数有几种：capacity_orig_of()返回最大capacity，capacity_of()返回减去rt占用的capacity，capacity_curr_of()返回当前频率下的最大capacity。

```
static inline unsigned long capacity_of(int cpu)
{
    return cpu_rq(cpu)->cpu_capacity;
}

static inline unsigned long capacity_orig_of(int cpu)
{
    return cpu_rq(cpu)->cpu_capacity_orig;
}

static inline unsigned long capacity_curr_of(int cpu)
{
    return cpu_rq(cpu)->cpu_capacity_orig *
           arch_scale_freq_capacity(NULL, cpu)
           >> SCHED_CAPACITY_SHIFT;
}
```

- __update_load_avg()函数完整的计算过程

```
#if (SCHED_LOAD_SHIFT - SCHED_LOAD_RESOLUTION) != 10 || SCHED_CAPACITY_SHIFT
!= 10
```

```
#error "load tracking assumes 2^10 as unit"
#endif

#define cap_scale(v, s) ((v)*(s) >> SCHED_CAPACITY_SHIFT)

/*
 * We can represent the historical contribution to runnable average as the
 * coefficients of a geometric series.  To do this we sub-divide our
runnable
 * history into segments of approximately 1ms (1024us); label the segment
that
 * occurred N-ms ago p_N, with p_0 corresponding to the current period, e.g.
 *
 * [<- 1024us ->|<- 1024us ->|<- 1024us ->| ...
 *      p0            p1           p2
 *     (now)       (~1ms ago)  (~2ms ago)
 *
 * Let u_i denote the fraction of p_i that the entity was runnable.
 *
 * We then designate the fractions u_i as our co-efficients, yielding the
 * following representation of historical load:
 *   u_0 + u_1*y + u_2*y^2 + u_3*y^3 + ...
 *
 * We choose y based on the with of a reasonably scheduling period, fixing:
 *   y^32 = 0.5
 *
 * This means that the contribution to load ~32ms ago (u_32) will be
weighted
 * approximately half as much as the contribution to load within the last ms
 * (u_0).
 *
 * When a period "rolls over" and we have new u_0`, multiplying the previous
 * sum again by y is sufficient to update:
 *   load_avg = u_0` + y*(u_0 + u_1*y + u_2*y^2 + ... )
 *            = u_0 + u_1*y + u_2*y^2 + ... [re-labeling u_i --> u_{i+1}]
 */
static __always_inline int
__update_load_avg(u64 now, int cpu, struct sched_avg *sa,
          unsigned long weight, int running, struct cfs_rq *cfs_rq)
{
    u64 delta, scaled_delta, periods;
    u32 contrib;
    unsigned int delta_w, scaled_delta_w, decayed = 0;
    unsigned long scale_freq, scale_cpu;

#ifdef CONFIG_64BIT_ONLY_CPU
    struct sched_entity *se;
    unsigned long load_avg_before = sa->load_avg;
#endif

    delta = now - sa->last_update_time;
    /*
     * This should only happen when time goes backwards, which it
     * unfortunately does during sched clock init when we swap over to TSC.
```

```
        */
       if ((s64)delta < 0) {
           sa->last_update_time = now;
           return 0;
       }

       /*
        * Use 1024ns as the unit of measurement since it's a reasonable
        * approximation of 1us and fast to compute.
        */
       delta >>= 10;
       if (!delta)
           return 0;
       sa->last_update_time = now;

       scale_freq = arch_scale_freq_capacity(NULL, cpu);
       scale_cpu = arch_scale_cpu_capacity(NULL, cpu);
       trace_sched_contrib_scale_f(cpu, scale_freq, scale_cpu);

       /* delta_w is the amount already accumulated against our next period */
       delta_w = sa->period_contrib;
       if (delta + delta_w >= 1024) {
           decayed = 1;

           /* how much left for next period will start over, we don't know yet
   */
           sa->period_contrib = 0;

           /*
            * Now that we know we're crossing a period boundary, figure
            * out how much from delta we need to complete the current
            * period and accrue it.
            */
           delta_w = 1024 - delta_w;
           scaled_delta_w = cap_scale(delta_w, scale_freq);
           if (weight) {
               sa->load_sum += weight * scaled_delta_w;
               if (cfs_rq) {
                   cfs_rq->runnable_load_sum +=
                           weight * scaled_delta_w;
               }
           }
           if (running)
               sa->util_sum += scaled_delta_w * scale_cpu;

           delta -= delta_w;

           /* Figure out how many additional periods this update spans */
           periods = delta / 1024;
           delta %= 1024;

           sa->load_sum = decay_load(sa->load_sum, periods + 1);
           if (cfs_rq) {
               cfs_rq->runnable_load_sum =
```

```
                    decay_load(cfs_rq->runnable_load_sum, periods + 1);
            }
            sa->util_sum = decay_load((u64)(sa->util_sum), periods + 1);

            /* Efficiently calculate \sum (1..n_period) 1024*y^i */
            contrib = __compute_runnable_contrib(periods);
            contrib = cap_scale(contrib, scale_freq);
            if (weight) {
                sa->load_sum += weight * contrib;
                if (cfs_rq)
                    cfs_rq->runnable_load_sum += weight * contrib;
            }
            if (running)
                sa->util_sum += contrib * scale_cpu;
        }

        /* Remainder of delta accrued against u_0` */
        scaled_delta = cap_scale(delta, scale_freq);
        if (weight) {
            sa->load_sum += weight * scaled_delta;
            if (cfs_rq)
                cfs_rq->runnable_load_sum += weight * scaled_delta;
        }
        if (running)
            sa->util_sum += scaled_delta * scale_cpu;

        sa->period_contrib += delta;

        if (decayed) {
            sa->load_avg = div_u64(sa->load_sum, LOAD_AVG_MAX);
            if (cfs_rq) {
                cfs_rq->runnable_load_avg =
                    div_u64(cfs_rq->runnable_load_sum, LOAD_AVG_MAX);
            }
            sa->util_avg = sa->util_sum / LOAD_AVG_MAX;
        }

    #ifdef CONFIG_64BIT_ONLY_CPU
        if (!cfs_rq) {
            if (is_sched_avg_32bit(sa)) {
                se = container_of(sa, struct sched_entity, avg);
                cfs_rq_of(se)->runnable_load_avg_32bit +=
                    sa->load_avg - load_avg_before;
            }
        }
    #endif

        return decayed;
    }
```

三、cpu级的负载计算update_cpu_load_active()

__update_load_avg()是计算se/cfs_rq级别的负载，在cpu级别linux使用update_cpu_load_active()来计算整个cpu->rq负载的变化趋势。计算也是周期性的，周期为1 tick。

```
scheduler_tick()

↓

void update_cpu_load_active(struct rq *this_rq)
{
    /* (1) 被累计的为：当前rqrunnable平均负载带weight分量
(cpu->rq->cfs_rq->runnable_load_avg) */
    unsigned long load = weighted_cpuload(cpu_of(this_rq));
    /*
     * See the mess around update_idle_cpu_load() / update_cpu_load_nohz().
     */
    this_rq->last_load_update_tick = jiffies;
    /* (2)   */
    __update_cpu_load(this_rq, load, 1);
}

|→

/*
 * Update rq->cpu_load[] statistics. This function is usually called every
 * scheduler tick (TICK_NSEC). With tickless idle this will not be called
 * every tick. We fix it up based on jiffies.
 */
static void __update_cpu_load(struct rq *this_rq, unsigned long this_load,
                  unsigned long pending_updates)
{
    int i, scale;

    this_rq->nr_load_updates++;

    /* Update our load: */
    /* (2.1) 逐个计算cpu_load[]中5个时间等级的值 */
    this_rq->cpu_load[0] = this_load; /* Fasttrack for idx 0 */
    for (i = 1, scale = 2; i < CPU_LOAD_IDX_MAX; i++, scale += scale) {
        unsigned long old_load, new_load;

        /* scale is effectively 1 << i now, and >> i divides by scale */

        old_load = this_rq->cpu_load[i];
        /* (2.2) 如果因为进入noHZ模式，有pending_updates个tick没有更新，
            先老化原有负载
         */
        old_load = decay_load_missed(old_load, pending_updates - 1, i);
        new_load = this_load;
        /*
         * Round up the averaging division if load is increasing. This
         * prevents us from getting stuck on 9 if the load is 10, for
         * example.
         */
        if (new_load > old_load)
```

```
            new_load += scale - 1;

        /* (2.3) cpu_load的计算公式 */
        this_rq->cpu_load[i] = (old_load * (scale - 1) + new_load) >> i;
    }

    sched_avg_update(this_rq);
}
```

代码注释中详细解释了cpu_load的计算方法：

- 每个tick计算不同idx时间等级的load，计算公式：load = (2^idx - 1) / 2^idx * load + 1 / 2^idx * cur_load
- 如果cpu因为noHZ错过了(n-1)个tick的更新，那么计算load要分两步：首先老化(decay)原有的load：load = ((2^idx - 1) / 2^idx)^(n-1) * load 再按照一般公式计算load：load = (2^idx - 1) / 2^idx) * load + 1 / 2^idx * cur_load
- 为了decay的加速计算，设计了decay_load_missed()查表法计算：

```
/*
 * The exact cpuload at various idx values, calculated at every tick would be
 * load = (2^idx - 1) / 2^idx * load + 1 / 2^idx * cur_load
 *
 * If a cpu misses updates for n-1 ticks (as it was idle) and update gets called
 * on nth tick when cpu may be busy, then we have:
 * load = ((2^idx - 1) / 2^idx)^(n-1) * load
 * load = (2^idx - 1) / 2^idx) * load + 1 / 2^idx * cur_load
 *
 * decay_load_missed() below does efficient calculation of
 * load = ((2^idx - 1) / 2^idx)^(n-1) * load
 * avoiding 0..n-1 loop doing load = ((2^idx - 1) / 2^idx) * load
 *
 * The calculation is approximated on a 128 point scale.
 * degrade_zero_ticks is the number of ticks after which load at any
 * particular idx is approximated to be zero.
 * degrade_factor is a precomputed table, a row for each load idx.
 * Each column corresponds to degradation factor for a power of two ticks,
 * based on 128 point scale.
 * Example:
 * row 2, col 3 (=12) says that the degradation at load idx 2 after
 * 8 ticks is 12/128 (which is an approximation of exact factor 3^8/4^8).
 *
 * With this power of 2 load factors, we can degrade the load n times
 * by looking at 1 bits in n and doing as many mult/shift instead of
 * n mult/shifts needed by the exact degradation.
 */
#define DEGRADE_SHIFT       7
static const unsigned char
        degrade_zero_ticks[CPU_LOAD_IDX_MAX] = {0, 8, 32, 64, 128};
static const unsigned char
        degrade_factor[CPU_LOAD_IDX_MAX][DEGRADE_SHIFT + 1] = {
                    {0, 0, 0, 0, 0, 0, 0, 0},
                    {64, 32, 8, 0, 0, 0, 0, 0},
                    {96, 72, 40, 12, 1, 0, 0},
                    {112, 98, 75, 43, 15, 1, 0},
```

```
                      {120, 112, 98, 76, 45, 16, 2} };

/*
 * Update cpu_load for any missed ticks, due to tickless idle. The backlog
 * would be when CPU is idle and so we just decay the old load without
 * adding any new load.
 */
static unsigned long
decay_load_missed(unsigned long load, unsigned long missed_updates, int idx)
{
    int j = 0;

    if (!missed_updates)
        return load;

    if (missed_updates >= degrade_zero_ticks[idx])
        return 0;

    if (idx == 1)
        return load >> missed_updates;

    while (missed_updates) {
        if (missed_updates % 2)
            load = (load * degrade_factor[idx][j]) >> DEGRADE_SHIFT;

        missed_updates >>= 1;
        j++;
    }
    return load;
}
```

- cpu_load[]含5条均线，反应不同时间窗口长度下的负载情况；主要供load_balance()在不 同场景判断是否负载平衡的比较基准，常用为cpu_load[0]和cpu_load[1];
- cpu_load[index]对应的时间长度为{0, 8, 32, 64, 128}，单位为tick;
- 移动均线的目的在于平滑样本的抖动，确定趋势的变化方向;

四、系统级的负载计算calc_global_load_tick()

系统级的平均负载(load average)可以通过以下命令(uptime、top、cat /proc/loadavg)查看：

```
$ uptime
 16:48:24 up  4:11,  1 user,  load average: 25.25, 23.40, 23.46

$ top - 16:48:42 up  4:12,  1 user,  load average: 25.25, 23.14, 23.37

$ cat /proc/loadavg
25.72 23.19 23.35 42/3411 43603
```

“load average:”后面的3个数字分别表示1分钟、5分钟、15分钟的load average。可以从几方面去解析load average：

- If the averages are 0.0, then your system is idle.
- If the 1 minute average is higher than the 5 or 15 minute averages, then load is increasing.
- If the 1 minute average is lower than the 5 or 15 minute averages, then load is decreasing.

- If they are higher than your CPU count, then you might have a performance problem (it depends).

最早的系统级平均负载(load average)只会统计runnable状态。但是linux后面觉得这种统计方式代表不了系统的真实负载；举一个例子：系统换一个低速硬盘后，他的runnable负载还会小于高速硬盘时的值；linux认为睡眠状态(TASK_INTERRUPTIBLE/TASK_UNINTERRUPTIBLE)也是系统的一种负载，系统得不到服务是因为io/外设的负载过重；系统级负载统计函数calc_global_load_tick()中会把(this_rq->nr_running+this_rq->nr_uninterruptible)都计入负载；

1.4.1 calc_global_load_tick()

我们来看详细的代码解析。

- 每个cpu每隔5s更新本cpu rq的(nr_running+nr_uninterruptible)任务数量到系统全局变量calc_load_tasks，calc_load_tasks是整系统多个cpu(nr_running+nr_uninterruptible)任务数量的总和，多cpu在访问calc_load_tasks变量时使用原子操作来互斥。

```
scheduler_tick()

↓

void calc_global_load_tick(struct rq *this_rq)
{
    long delta;

    /* (1) 5S的更新周期 */
    if (time_before(jiffies, this_rq->calc_load_update))
        return;

    /* (2) 计算本cpu的负载变化到全局变量calc_load_tasks中 */
    delta  = calc_load_fold_active(this_rq);
    if (delta)
        atomic_long_add(delta, &calc_load_tasks);

    this_rq->calc_load_update += LOAD_FREQ;
}
```

- 多个cpu更新calc_load_tasks，但是计算load只由一个cpu来完成，这个cpu就是tick_do_timer_cpu。在linux time一文中，我们看到这个cpu就是专门来更新时间戳timer的(update_wall_time())。实际上它在更新时间戳的同时也会调用do_timer() -> calc_global_load()来计算系统负载。核心算法calc_load()的思想也是：旧的load*老化系数 + 新load*系数。假设单位1为FIXED_1=2^11=2028，EXP_1=1884、EXP_5=2014、EXP_15=2037，load的计算：
  load = old_load*(EXP_?/FIXED_1) + new_load*(FIXED_1-EXP_?)/FIXED_1

```
do_timer() -> calc_global_load()

↓

void calc_global_load(unsigned long ticks)
{
    long active, delta;

    /* (1) 计算的间隔时间为5s + 10tick，
```

```
        加10tick的目的就是让所有cpu都更新完calc_load_tasks，
        tick_do_timer_cpu再来计算
     */
    if (time_before(jiffies, calc_load_update + 10))
        return;

    /*
     * Fold the 'old' idle-delta to include all NO_HZ cpus.
     */
    delta = calc_load_fold_idle();
    if (delta)
        atomic_long_add(delta, &calc_load_tasks);

    /* (2) 读取全局统计变量 */
    active = atomic_long_read(&calc_load_tasks);
    active = active > 0 ? active * FIXED_1 : 0;

    /* (3) 计算1分钟、5分钟、15分钟的负载 */
    avenrun[0] = calc_load(avenrun[0], EXP_1, active);
    avenrun[1] = calc_load(avenrun[1], EXP_5, active);
    avenrun[2] = calc_load(avenrun[2], EXP_15, active);

    calc_load_update += LOAD_FREQ;

    /*
     * In case we idled for multiple LOAD_FREQ intervals, catch up in bulk.
     */
    calc_global_nohz();
}

|→

/*
 * a1 = a0 * e + a * (1 - e)
 */
static unsigned long
calc_load(unsigned long load, unsigned long exp, unsigned long active)
{
    unsigned long newload;

    newload = load * exp + active * (FIXED_1 - exp);
    if (active >= load)
        newload += FIXED_1-1;

    return newload / FIXED_1;
}

#define FSHIFT      11      /* nr of bits of precision */
#define FIXED_1     (1<<FSHIFT) /* 1.0 as fixed-point */
#define LOAD_FREQ   (5*HZ+1)    /* 5 sec intervals */
#define EXP_1       1884        /* 1/exp(5sec/1min) as fixed-point */
#define EXP_5       2014        /* 1/exp(5sec/5min) */
#define EXP_15      2037        /* 1/exp(5sec/15min) */
```

- /proc/loadavg。代码实现在kernel/fs/proc/loadavg.c中

```c
#define LOAD_INT(x) ((x) >> FSHIFT)
#define LOAD_FRAC(x) LOAD_INT(((x) & (FIXED_1-1)) * 100)

static int loadavg_proc_show(struct seq_file *m, void *v)
{
    unsigned long avnrun[3];

    get_avenrun(avnrun, FIXED_1/200, 0);

    seq_printf(m, "%lu.%02lu %lu.%02lu %lu.%02lu %ld/%d %d\n",
        LOAD_INT(avnrun[0]), LOAD_FRAC(avnrun[0]),
        LOAD_INT(avnrun[1]), LOAD_FRAC(avnrun[1]),
        LOAD_INT(avnrun[2]), LOAD_FRAC(avnrun[2]),
        nr_running(), nr_threads,
        task_active_pid_ns(current)->last_pid);
    return 0;
}
```

五、占用率统计

1.5.1、cputime.c：top命令利用“/proc/stat”、“/proc/stat”来做cpu占用率统计，可以在AOSP/system/core/toolbox/top.c中查看top代码实现。读取/proc/stat可以查看系统各种状态的时间统计，代码实现在fs/proc/stat.c show_stat()。

```
# cat /proc/stat

/* 系统时间的累加，格式 =
"cpu, user, nice, system, idle, iowait, irq, softirq, steal, guest, guest_nice"
*/
cpu  4022747 54885 15739405 106716492 190413 0 38250 0 0 0
cpu0 2507238 44342 9881429 87084619 154904 0 32594 0 0 0
intr 242500437 0 0 149757888 0 0 5 15529 0 0 0 0 0 0 0 0 18385 3111402 0
6862026 128 0 0 2 0 0 0 0 276502 2317633 4713710 0 0 0 3 2604 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1211 0 760467 0 0 27713 0 0 0 0 0 0 0 0 0 0 0
0 1789515 8333417 1369 3344 2399 389 0 23665294 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 4068 0 0 0 0 13984 0 0 0 0 0 0 0 0 0 0 0 13585 169 13590 169 19300 169 0 0
0 0 0 45 0 9622 0 0 0 0 0 0 0 0 0 27026 1948 0 19475 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 242 0 0 0 0 0 0 0 2580 2595 0 0 0 873 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 2 0 0 17 0 0 0 0 1
7340 0 0 635 102644 167 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

```
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 4
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 292 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 58 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0

/* 格式：
nctxt: sum += cpu_rq(i)->nr_switches;
btime：boottime.tv_sec
processes：total_forks
procs_running：sum += cpu_rq(i)->nr_running;
procs_blocked：sum += atomic_read(&cpu_rq(i)->nr_iowait);
*/
ctxt 384108244
btime 1512114477
processes 130269
procs_running 1
procs_blocked 0

/* 软中断的次数统计，格式：
softirq, 全局统计, HI_SOFTIRQ, TIMER_SOFTIRQ, NET_TX_SOFTIRQ, NET_RX_SOFTIRQ,
BLOCK_SOFTIRQ, BLOCK_IOPOLL_SOFTIRQ, TASKLET_SOFTIRQ, SCHED_SOFTIRQ,
HRTIMER_SOFTIRQ, RCU_SOFTIRQ
*/
softirq 207132697 736178 121273868 735555 9094 2134399 734917 746032
14491717 0 66270937
```

读取/proc/pid/stat可以查看进程各种状态的时间统计，代码实现在fs/proc/array.c do_task_stat()。

```
# cat /proc/824/stat
824 (ifaad) S 1 824 0 0 -1 4210944 600 0 2 0 1 1 0 0 20 0 1 0 2648 12922880
1066 18446744073709551615 416045604864 416045622068 548870218464
548870217760 500175854100 0 0 0 32768 1 0 0 17 6 0 0 0 0 0 416045629200
416045633544 416159543296 548870220457 548870220475 548870220475
548870221798 0
```

相关的时间统计是在cputime.c中实现的，在每个tick任务中通过采样法计算系统和进程不同状态下的时间统计，这种方法精度是不高的：

- 采样法只能在tick时采样，中间发生了任务调度不可统计；
- 系统统计了以下几种类型：

```
enum cpu_usage_stat {
    CPUTIME_USER,
    CPUTIME_NICE,
    CPUTIME_SYSTEM,
    CPUTIME_SOFTIRQ,
    CPUTIME_IRQ,
    CPUTIME_IDLE,
    CPUTIME_IOWAIT,
    CPUTIME_STEAL,
    CPUTIME_GUEST,
    CPUTIME_GUEST_NICE,
    NR_STATS,
};
```

- 在nohz模式时，退出nohz时会使用tick_nohz_idle_exit() -> tick_nohz_account_idle_ticks() -> account_idle_ticks()加上nohz损失的idle时间；tick统计的代码详细解析如下：

```
update_process_times() -> account_process_tick()

↓

void account_process_tick(struct task_struct *p, int user_tick)
{
    cputime_t one_jiffy_scaled = cputime_to_scaled(cputime_one_jiffy);
    struct rq *rq = this_rq();

    if (vtime_accounting_enabled())
        return;

    if (sched_clock_irqtime) {
        /* (1) 如果irq的时间需要被统计，使用新的函数 */
        irqtime_account_process_tick(p, user_tick, rq, 1);
        return;
    }

    if (steal_account_process_tick())
        return;

    if (user_tick)
        /* (2) 统计用户态时间 */
        account_user_time(p, cputime_one_jiffy, one_jiffy_scaled);
    else if ((p != rq->idle) || (irq_count() != HARDIRQ_OFFSET))
        /* (3) 统计用户态时间 */
        account_system_time(p, HARDIRQ_OFFSET, cputime_one_jiffy,
                    one_jiffy_scaled);
    else
        /* (4) 统计idle时间 */
        account_idle_time(cputime_one_jiffy);
}

|→

static void irqtime_account_process_tick(struct task_struct *p, int
user_tick,
                        struct rq *rq, int ticks)
{
    /* (1.1) 1 tick的时间 */
    cputime_t scaled = cputime_to_scaled(cputime_one_jiffy);
    u64 cputime = (__force u64) cputime_one_jiffy;

    /* (1.2) cpu级别的统计结构：kcpustat_this_cpu->cpustat */
    u64 *cpustat = kcpustat_this_cpu->cpustat;

    if (steal_account_process_tick())
        return;

    cputime *= ticks;
    scaled *= ticks;
```

```

    /* (1.3) 如果irq时间已经增加，把本tick 时间加到IRQ时间，加入cpu级别统计 */
    if (irqtime_account_hi_update()) {
        cpustat[CPUTIME_IRQ] += cputime;

    /* (1.4) 如果softirq时间已经增加，把本tick 时间加到SOFTIRQ时间，加入cpu级别统计
*/
    } else if (irqtime_account_si_update()) {
        cpustat[CPUTIME_SOFTIRQ] += cputime;

    /* (1.5) 加入system内核态 CPUTIME_SOFTIRQ时间，加入cpu级别、进程级别统计 */
    } else if (this_cpu_ksoftirqd() == p) {
        /*
         * ksoftirqd time do not get accounted in cpu_softirq_time.
         * So, we have to handle it separately here.
         * Also, p->stime needs to be updated for ksoftirqd.
         */
        __account_system_time(p, cputime, scaled, CPUTIME_SOFTIRQ);

    /* (1.6) 加入用户态时间，加入cpu级别、进程级别统计 */
    } else if (user_tick) {
        account_user_time(p, cputime, scaled);

    /* (1.7) 加入idle时间，加入cpu级别统计 */
    } else if (p == rq->idle) {
        account_idle_time(cputime);

    /* (1.8) 加入guest时间，把system时间转成user时间 */
    } else if (p->flags & PF_VCPU) { /* System time or guest time */
        account_guest_time(p, cputime, scaled);

    /* (1.9) 加入system内核态 CPUTIME_SYSTEM时间，加入cpu级别、进程级别统计 */
    } else {
        __account_system_time(p, cputime, scaled,   CPUTIME_SYSTEM);
    }
}

||→

static inline
void __account_system_time(struct task_struct *p, cputime_t cputime,
            cputime_t cputime_scaled, int index)
{
    /* Add system time to process. */
    /* (1.5.1) 增加进程级别的内核态时间p->stime */
    p->stime += cputime;
    p->stimescaled += cputime_scaled;
    /* 统计task所在线程组(thread group)的运行时间：
tsk->signal->cputimer->cputime_atomic.stime */
    account_group_system_time(p, cputime);

    /* Add system time to cpustat. */
    /* (1.5.2) 更新CPU级别的cpustat统计：kernel_cpustat.cpustat[index]
        更新cpuacct的cpustat统计：ca->cpustat->cpustat[index]
```

```
     */
    task_group_account_field(p, index, (__force u64) cputime);

    /* Account for system time used */
    /* (1.5.3) 更新tsk->acct_timexpd、tsk->acct_rss_mem1、tsk->acct_vm_mem1 */
    acct_account_cputime(p);
}

||→

void account_user_time(struct task_struct *p, cputime_t cputime,
               cputime_t cputime_scaled)
{
    int index;

    /* Add user time to process. */
    /* (1.6.1) 增加进程级别的用户态时间p->utime */
    p->utime += cputime;
    p->utimescaled += cputime_scaled;
    /* 统计task所在线程组(thread group)的运行时间：
tsk->signal->cputimer->cputime_atomic.utime */
    account_group_user_time(p, cputime);

    index = (task_nice(p) > 0) ? CPUTIME_NICE : CPUTIME_USER;

    /* Add user time to cpustat. */
    /* (1.6.2) 更新CPU级别的cpustat统计：kernel_cpustat.cpustat[index]
        更新cpuacct的cpustat统计：ca->cpustat->cpustat[index]
     */
    task_group_account_field(p, index, (__force u64) cputime);

    /* Account for user time used */
    /* (1.6.3) 更新tsk->acct_timexpd、tsk->acct_rss_mem1、tsk->acct_vm_mem1 */
    acct_account_cputime(p);
}

||→

void account_idle_time(cputime_t cputime)
{
    u64 *cpustat = kcpustat_this_cpu->cpustat;
    struct rq *rq = this_rq();

    /* (1.7.1) 把本tick 时间加到CPUTIME_IOWAIT时间，加入cpu级别统计 */
    if (atomic_read(&rq->nr_iowait) > 0)
        cpustat[CPUTIME_IOWAIT] += (__force u64) cputime;

    /* (1.7.1) 把本tick 时间加到CPUTIME_IDLE时间，加入cpu级别统计 */
    else
        cpustat[CPUTIME_IDLE] += (__force u64) cputime;
}

||→

```

```c
static void account_guest_time(struct task_struct *p, cputime_t cputime,
                   cputime_t cputime_scaled)
{
    u64 *cpustat = kcpustat_this_cpu->cpustat;

    /* Add guest time to process. */
    p->utime += cputime;
    p->utimescaled += cputime_scaled;
    account_group_user_time(p, cputime);
    p->gtime += cputime;

    /* Add guest time to cpustat. */
    if (task_nice(p) > 0) {
        cpustat[CPUTIME_NICE] += (__force u64) cputime;
        cpustat[CPUTIME_GUEST_NICE] += (__force u64) cputime;
    } else {
        cpustat[CPUTIME_USER] += (__force u64) cputime;
        cpustat[CPUTIME_GUEST] += (__force u64) cputime;
    }
}
```