调度器介绍

一个好的调度算法应当考虑以下几个方面：

- **公平**：保证每个进程得到合理的CPU时间。
- **高效**：使CPU保持忙碌状态，即总是有进程在CPU上运行。
- **响应时间**：使交互用户的响应时间尽可能短。
- **周转时间**：使批处理用户等待输出的时间尽可能短。
- **吞吐量**：使单位时间内处理的进程数量尽可能多。
- **负载均衡**：在多核多处理器系统中提供更高的性能

而整个调度系统至少包含两种调度算法，是分别针对**实时进程**和**普通进程**，所以在整个linux内核中，实时进程和普通进程是并存的，但它们使用的调度算法并不相同，普通进程使用的是CFS调度算法(红黑树调度)。之后会介绍调度器是怎么调度这两种进程。

进程

上面已经说明，在linux中，进程主要分为两种，一种为实时进程，一种为普通进程

- **实时进程**：对系统的响应时间要求很高，它们需要短的响应时间，并且这个时间的变化非常小，典型的实时进程有音乐播放器，视频播放器等。
- **普通进程**：包括交互进程和非交互进程，交互进程如文本编辑器，它会不断的休眠，又不断地通过鼠标键盘进行唤醒，而非交互进程就如后台维护进程，他们对IO，响应时间没有很高的要求，比如编译器。

它们在linux内核运行时是共存的，实时进程的优先级为0~99，实时进程优先级不会在运行期间改变(静态优先级)，而普通进程的优先级为100~139，普通进程的优先级会在内核运行期间进行相应的改变(动态优先级)。

调度策略

在linux系统中，调度策略分为

- **SCHED_NORMAL**：普通进程使用的调度策略，现在此调度策略使用的是CFS调度器。
- **SCHED_FIFO**：实时进程使用的调度策略，此调度策略的进程一旦使用CPU则一直运行，直到有比其更高优先级的实时进程进入队列，或者其自动放弃CPU，适用于时间性要求比较高，但每次运行时间比较短的进程。
- **SCHED_RR**：实时进程使用的时间片轮转法策略，实时进程的时间片用完后，调度器将其放到队列末尾，这样每个实时进程都可以执行一段时间。适用于每次运行时间比较长的实时进程。

调度

首先，我们需要清楚，什么样的进程会进入调度器进行选择，就是处于TASK_RUNNING状态的进程，而其他状态下的进程都不会进入调度器进行调度。系统发生调度的时机如下

- 调用cond_resched()时
- 显式调用schedule()时
- 从系统调用或者异常中断返回用户空间时
- 从中断上下文返回用户空间时

当开启**内核抢占(默认开启)**时，会多出几个调度时机，如下

- 在系统调用或者异常中断上下文中调用preempt_enable()时(多次调用preempt_enable()时，系统只会在最后一次调用时会调度)
- 在中断上下文中，从中断处理函数返回到可抢占的上下文时(这里是中断下半部，中断上半部实际上会关中断，而新的中断只会被登记，由于上半部处理很快，上半部处理完成后才会执行新的中断信号，这样就形成了中断可重入)

而在系统启动调度器初始化时会初始化一个调度定时器，调度定时器每隔一定时间执行一个中断，在中断会对当前运行进程运行时间进行更新，如果进程需要被调度，在调度定时器中断中会设置一个调度标志位，之后从定时器中断返回，因为上面已经提到从中断上下文返回时是有调度时机的，在内核源码的汇编代码中所有中断返回处理都必须去判断调度标志位是否设置，如设置则执行schedule()进行调度。而我们知道实时进程和普通进程是共存的，调度器是怎么协调它们之间的调度的呢，其实很简单，每次调度时，会先在实时进程运行队列中查看是否有可运行的实时进程，如果没有，再去普通进程运行队列找下一个可运行的普通进程，如果也没有，则调度器会使用idle进程进行运行。之后的章节会放上代码进行详细说明。

系统并不是每时每刻都允许调度的发生，当处于硬中断期间的时候，调度是被系统禁止的，之后硬中断过后才重新允许调度。而对于异常，系统并不会禁止调度，也就是在异常上下文中，系统是有可能发生调度的。

数据结构

在这一节中，我们都是以普通进程作为讲解对象，因为普通进程使用的调度算法为CFS调度算法，它是以红黑树为基础的调度算法，其相比与实时进程的调度算法复杂很多，而实时进程在组织结构上与普通进程没有太大差别，算法也较为简单。

组成形式

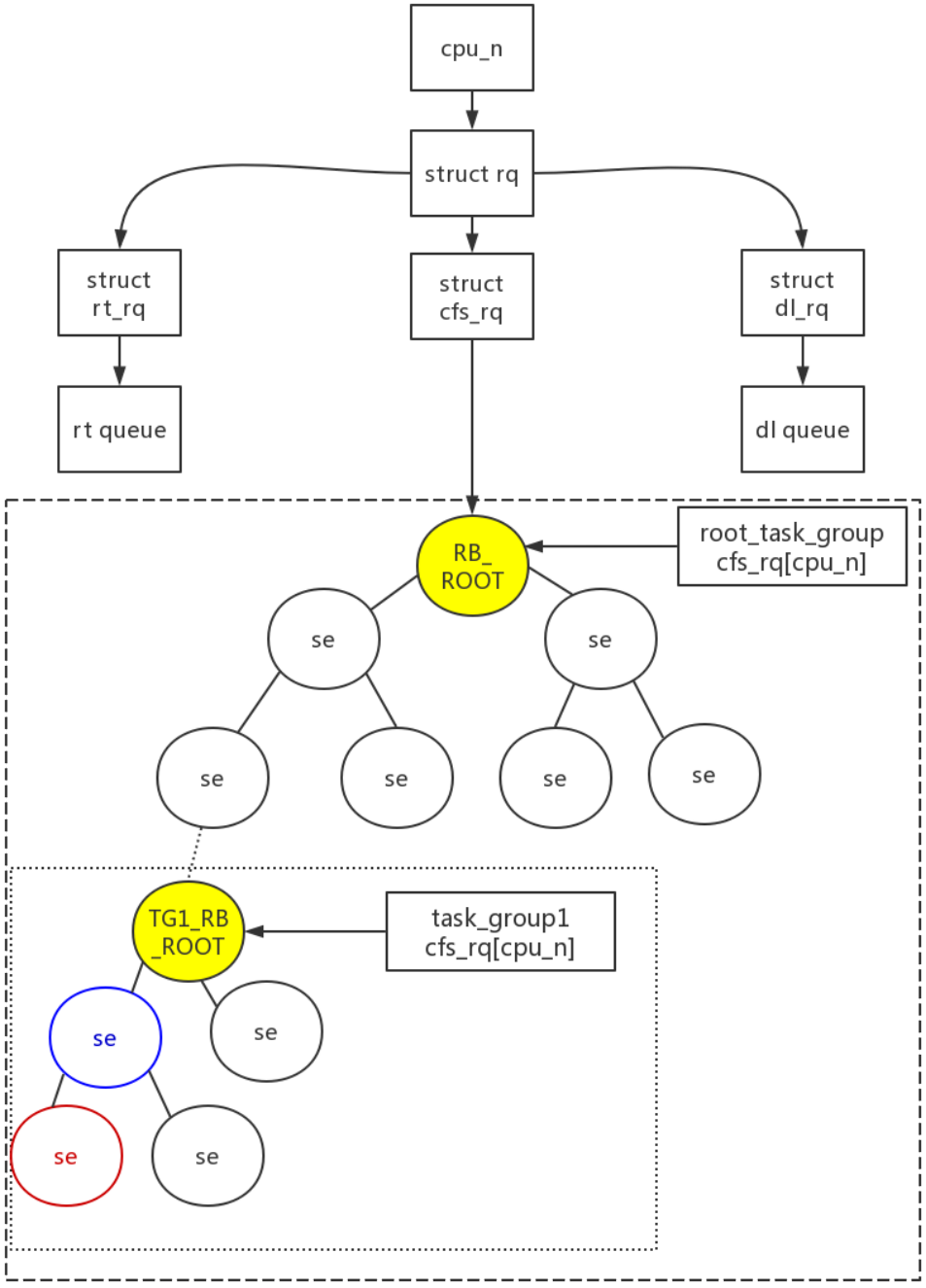


图1

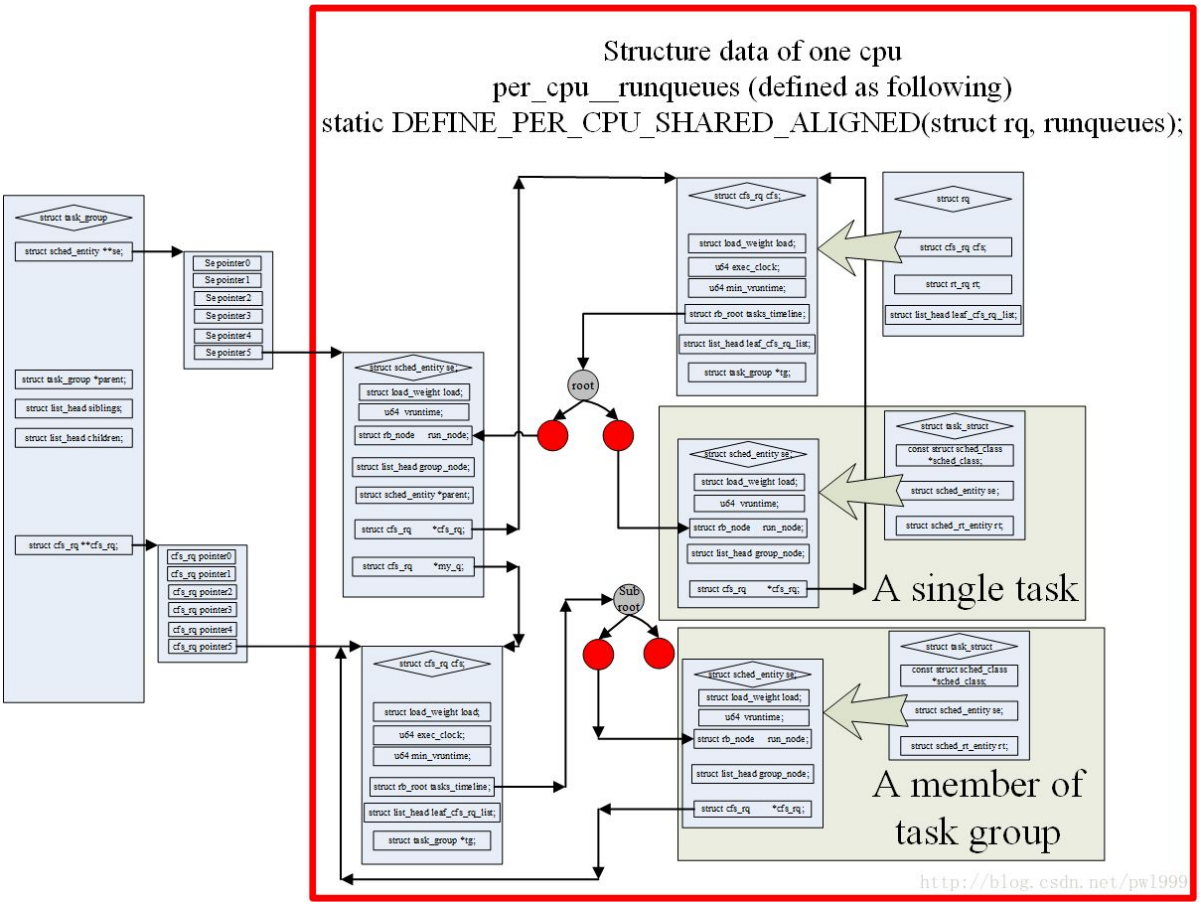
从这张图片中，来了解task_group,sched_entity,cfs_rq，root_task_group的关系。

- 每一个cpu都一个rq
- 每一个rq里面都包含了cfs_rq/rt_rq/dl_rq，cfs_rq是普通进程rq，rt_rq是实时进程rq，dl_rq目前没有搞清楚，以后在研究，上图以cfs_rq示意图分析
- se是sched_entity是调度实体，可能为一个task，可能为task_group，即进程组，包含多个task，而且这个任务组内的每个task可以运行在不同的CPU mask上
- 如果调度实体仅仅是一个task，由于根节点root_task_group来组织的，也即每个cpu上的cfs rq都以root_task_group来组织RB tree，节点就是调度实体(单个任务/任务组)。

单个task组成的进程组，调度器直接根据rb tree选择最左边的叶子节点作为调度实体放入运行队列中运行

- 如果调度实体是task group，进程组作为一个调度实体存在，它包含若干个task，并且若干个task可能不在相同的cpu上运行，那么这个调度组就相当于一个自成一体的rb tree，跟基本的root_task_group一样，组成一个rb tree。比如图中所示的TG1_RB_ROOT作为这个组调度的跟节点，实际还是挂载在其他调度实体的下面。
- 调度器在pick task的时候，如果是进程组，会进入到TG1_RB_ROOT中，寻找rb tree最左边的叶子节点进行调度。
- 由于在多cpu情况下，在一个进程组的进程有可能在不同的cpu上同时运行，所以每个进程组都必须对每个cpu分配它的调度实体(struct sched_entity/struct sched_rt_entity)和运行队列(struct cfs_rq/struct rt_rq)。这个初始化过程后面在讲解。
- 比如上图中的红色球形se，这个调度实体正在此cpu上运行，运行时此红色se会从rb tree上剥离处理，运行完毕之后在重新计算其插入rb tree的位置并pick next task运行，即下一个蓝色的调度实体se(task_group1内)。否则挑选其他调度实体。

下面这张图更加形象的说明了一个调度实体与进程组的关系，看my_q这个参数的桥接作用：



下面来分别来讲解几个重要的结构体：

组调度(struct task_group)

我们知道，linux是一个多用户系统，如果有两个进程分别属于两个用户，而进程的优先级不同，会导致两个用户所占用的CPU时间不同，这样显然是不公平的(如果优先级差距很大，低优先级进程所属用户使用CPU的时间就很小)，所以内核引入组调度。如果基于用户分组，即使进程优先级不同，这两个用户使用的CPU时间都为50%。这就是为什么图1 中CPU0有两个蓝色将被调度的程序，如果task_group1中的运行时间还没有使用完，而当前进程运行时间使用完后，会调度

task_group1中的下一个被调度进程；相反，如果task_group1的运行时间使用结束，则调用上一层的下一个被调度进程。需要注意的是，一个组调度中可能会有一部分是实时进程，一部分是普通进程，这也导致这种组要能够满足即能在实时调度中进行调度，又可以在CFS调度中进行调度。

linux可以以以下两种方式进行进程的分组：

- **用户ID**：按照进程的USER ID进行分组，在对应的/sys/kernel/uid/目录下会生成一个cpu.share的文件，可以通过配置该文件来配置用户所占CPU时间比例
- **cgourp(control group)**：生成组用于限制其所有进程，比如我生成一个组(生成后此组为空，里面没有进程)，设置其CPU使用率为10%，并把一个进程丢进这个组中，那么这个进程最多只能使用CPU的10%，如果我们将多个进程丢进这个组，这个组的所有进程平分这个10%。

注意的是，这里的进程组概念和fork调用所产生的父子进程组概念不一样，文章所使用的进程组概念全为组调度中进程组的概念。为了管理组调度，内核引进了struct task_group结构，如下：

```
/* task group related information */
struct task_group {
    /* 用于进程找到其所属进程组结构 */
     struct cgroup_subsys_state css;

#ifdef CONFIG_FAIR_GROUP_SCHED
/* CFS调度器的进程组变量，在 alloc_fair_sched_group() 中进程初始化及分配内存 */
     /* 该进程组在每个CPU上都有对应的一个调度实体，因为有可能此进程组同时在两个CPU上
      运行(它的A进程在CPU0上运行，B进程在CPU1上运行) */
     /* schedulable entities of this group on each cpu */
     struct sched_entity **se;
     /* runqueue "owned" by this group on each cpu */
   /* 进程组在每个CPU上都有一个CFS运行队列(为什么需要，稍后解释) */
     struct cfs_rq **cfs_rq;
    /* 用于保存优先级默认为NICE 0的优先级 ，这个数值是常量*/
     unsigned long shares;

#ifdef  CONFIG_SMP
     atomic_long_t load_avg;
#endif
#endif

#ifdef CONFIG_RT_GROUP_SCHED
      /* 实时进程调度器的进程组变量，同 CFS */
     struct sched_rt_entity **rt_se;
     struct rt_rq **rt_rq;

     struct rt_bandwidth rt_bandwidth;
#endif

     struct rcu_head rcu;
      /* 用于建立进程链表(属于此调度组的进程链表) */
     struct list_head list;
     /* 指向其上层的进程组，每一层的进程组都是它上一层进程组的运行队列的一个调度实体
     ，在同一层中，进程组和进程被同等对待 */
     struct task_group *parent;
     /* 进程组的兄弟结点链表 */
     struct list_head siblings;
     /* 进程组的儿子结点链表 */
     struct list_head children;
```

```
#ifdef CONFIG_SCHED_AUTOGROUP
    struct autogroup *autogroup;
#endif

    struct cfs_bandwidth cfs_bandwidth;
};
```

在struct task_group结构中，最重要的成员为 struct sched_entity ** se 和 struct cfs_rq ** cfs_rq 。在图1 中，root_task_group与task_group1都只有一个，它们在初始化时会根据CPU个数为se和cfs_rq分配空间，即在task_group1和root_task_group中会为每个CPU分配一个se和cfs_rq，同理用于实时进程的 struct sched_rt_entity ** rt_se 和 struct rt_rq ** rt_rq也是一样。为什么这样呢，原因就是在多核多CPU的情况下，同一进程组的进程有可能在不同CPU上同时运行，所以每个进程组都必须对每个CPU分配它的调度实体(struct sched_entity 和 struct sched_rt_entity)和运行队列(struct cfs_rq 和 struct rt_rq)。怎么初始化的，后面会讲解。

调度实体(struct sched_entity)

在组调度中，也涉及到调度实体这个概念，它的结构为struct sched_entity(简称se)，就是图1红黑树中的se。其实际上就代表了一个调度对象，可以为一个进程，也可以为一个进程组。对于根的红黑树而言，一个进程组就相当于一个调度实体，一个进程也相当于一个调度实体。我们可以先看看其结构，如下：

```
/* 一个调度实体(红黑树的一个结点)，其包含一组或一个指定的进程，包含一个自己的运行队列，一个父亲指针，一个指向需要调度的运行队列指针 */
struct sched_entity {
    /* 权重，在数组prio_to_weight[]包含优先级转权重的数值 */
    struct load_weight    load;         /* for load-balancing */
    /* 实体在红黑树对应的结点信息 */
    struct rb_node        run_node;
    /* 实体所在的进程组 */
    struct list_head      group_node;
    /* 实体是否处于红黑树运行队列中 */
    unsigned int          on_rq;

    /* 开始运行时间 */
    u64                   exec_start;
    /* 总运行时间 */
    u64                   sum_exec_runtime;
    /* 虚拟运行时间，在时间中断或者任务状态发生改变时会更新
     * 其会不停增长，增长速度与load权重成反比，load越高，增长速度越慢，
       就越可能处于红黑树最左边被调度
     * 每次时钟中断都会修改其值
     * 具体见calc_delta_fair()函数
     */
    u64                   vruntime;
    /* 进程在切换进CPU时的sum_exec_runtime值 */
    u64                   prev_sum_exec_runtime;

    /* 此调度实体中进程移到其他CPU组的数量 */
    u64                   nr_migrations;

#ifdef CONFIG_SCHEDSTATS
    /* 用于统计一些数据 */
```

```
        struct sched_statistics statistics;
    #endif

    #ifdef CONFIG_FAIR_GROUP_SCHED
        /* 代表此进程组的深度，每个进程组都比其parent调度组深度大1 */
        int             depth;
        /* 父亲调度实体指针，如果是进程则指向其运行队列的调度实体，如果是进程组则指
            向其上一个进程组的调度实体
         * 在 set_task_rq 函数中设置
         */
        struct sched_entity     *parent;
        /* 实体所处红黑树运行队列 */
        struct cfs_rq           *cfs_rq;
        /* 实体的红黑树运行队列，如果为NULL表明其是一个进程，若非NULL表明其是调度组 */
        struct cfs_rq           *my_q;
    #endif

    #ifdef CONFIG_SMP
        /* Per-entity load-tracking */
        struct sched_avg    avg;
    #endif
    };
```

实际上，红黑树是根据 struct rb_node 建立起关系的，不过 struct rb_node 与 struct sched_entity 是一一对应关系，也可以简单看为一个红黑树结点就是一个调度实体。可以看出，在 struct sched_entity 结构中，包含了一个进程(或进程组)调度的全部数据，其被包含在 struct task_struct 结构中的se中，如下：

```
    struct task_struct {
        ........
        /* 表示是否在运行队列 */
        int on_rq;

        /* 进程优先级
         * prio: 动态优先级，范围为100~139，与静态优先级和补偿(bonus)有关
         * static_prio: 静态优先级，static_prio = 100 + nice + 20
           (nice值为-20~19,所以static_prio值为100~139)
         * normal_prio: 没有受优先级继承影响的常规优先级，具体见normal_prio函数
           ，跟属于什么类型的进程有关
         */
        int prio, static_prio, normal_prio;
        /* 实时进程优先级 */
        unsigned int rt_priority;
        /* 调度类，调度处理函数类 */
        const struct sched_class *sched_class;
        /* 调度实体(红黑树的一个结点) */
        struct sched_entity se;
        /* 调度实体(实时调度使用) */
        struct sched_rt_entity rt;
    #ifdef CONFIG_CGROUP_SCHED
        /* 指向其所在进程组 */
        struct task_group *sched_task_group;
    #endif
        ........
    }
```

在 struct sched_entity 结构中，值得我们注意的成员是：

- **load**：权重，通过优先级转换而成，是vruntime计算的关键。
- **on_rq**：表明是否处于CFS红黑树运行队列中，需要明确一个观点就是，CFS运行队列里面包含有一个红黑树，但这个红黑树并不是CFS运行队列的全部，因为红黑树仅仅是用于选择出下一个调度程序的算法。很简单的一个例子，普通程序运行时，其并不在红黑树中，但是还是处于CFS运行队列中，其on_rq为真。只有准备退出、即将睡眠等待和转为实时进程的进程其CFS运行队列的on_rq为假。
- **vruntime**：虚拟运行时间，调度的关键，其计算公式：一次调度间隔的虚拟运行时间 = 实际运行时间 * (NICE_0_LOAD / 权重)。可以看出跟实际运行时间和权重有关，红黑树就是以此作为排序的标准，优先级越高的进程在运行时其vruntime增长的越慢，其可运行时间相对就长，而且也越有可能处于红黑树的最左结点，调度器每次都选择最左边的结点为下一个调度进程。注意其值为单调递增，在每个调度器的时钟中断时当前进程的虚拟运行时间都会累加。单纯的说就是进程们都在比谁的vruntime最小，最小的将被调度。
- **cfs_rq**：此调度实体所处于的CFS运行队列。
- **my_q**：如果此调度实体代表的是一个进程组，那么此调度实体就包含有一个自己的CFS运行队列，其CFS运行队列中存放的是此进程组中的进程，这些进程就不会在其他CFS运行队列的红黑树中被包含(包括顶层红黑树也不会包含他们，他们只属于这个进程组的红黑树)。

对于怎么理解一个进程组有它自己的CFS运行队列，其实很好理解，比如在根CFS运行队列的红黑树上有一个进程A一个进程组B，各占50%的CPU，对于根的红黑树而言，他们就是两个调度实体。调度器调度的不是进程A就是进程组B，而如果调度到进程组B，进程组B自己选择一个程序交给CPU运行就可以了，而进程组B怎么选择一个程序给CPU，就是通过自己的CFS运行队列的红黑树选择，如果进程组B还有个子进程组C，原理都一样，就是一个层次结构。

而在 struct task_struct 结构中，我们注意到有个**调度类**，里面包含的是调度处理函数，它具体如下：

```
struct sched_class {
    /* 下一优先级的调度类
     * 调度类优先级顺序: stop_sched_class -> dl_sched_class ->
       rt_sched_class -> fair_sched_class -> idle_sched_class
     */
    const struct sched_class *next;

    /* 将进程加入到运行队列中，即将调度实体（进程）放入红黑树中，并对 nr_running
    变量加1 */
    void (*enqueue_task) (struct rq *rq, struct task_struct *p, int flags);
    /* 从运行队列中删除进程，并对 nr_running 变量中减1 */
    void (*dequeue_task) (struct rq *rq, struct task_struct *p, int flags);
    /* 放弃CPU，在 compat_yield sysctl 关闭的情况下，该函数实际上执行先出队后入队；在
这种情况下，它将调度实体放在红黑树的最右端 */
    void (*yield_task) (struct rq *rq);
    bool (*yield_to_task) (struct rq *rq, struct task_struct *p, bool
preempt);

    /* 检查当前进程是否可被新进程抢占 */
```

```
        void (*check_preempt_curr) (struct rq *rq, struct task_struct *p, int
    flags);

        /*
         * It is the responsibility of the pick_next_task() method that will
         * return the next task to call put_prev_task() on the @prev task or
         * something equivalent.
         *
         * May return RETRY_TASK when it finds a higher prio class has runnable
         * tasks.
         */
        /* 选择下一个应该要运行的进程运行 */
        struct task_struct * (*pick_next_task) (struct rq *rq,
                                struct task_struct *prev);
        /* 将进程放回运行队列 */
        void (*put_prev_task) (struct rq *rq, struct task_struct *p);

    #ifdef CONFIG_SMP
        /* 为进程选择一个合适的CPU */
        int (*select_task_rq)(struct task_struct *p, int task_cpu, int sd_flag,
    int flags);
        /* 迁移任务到另一个CPU */
        void (*migrate_task_rq)(struct task_struct *p, int next_cpu);
        /* 用于上下文切换后 */
        void (*post_schedule) (struct rq *this_rq);
        /* 用于进程唤醒 */
        void (*task_waking) (struct task_struct *task);
        void (*task_woken) (struct rq *this_rq, struct task_struct *task);
        /* 修改进程的CPU亲和力(affinity) */
        void (*set_cpus_allowed)(struct task_struct *p,
                         const struct cpumask *newmask);
        /* 启动运行队列 */
        void (*rq_online)(struct rq *rq);
        /* 禁止运行队列 */
        void (*rq_offline)(struct rq *rq);
    #endif
        /* 当进程改变它的调度类或进程组时被调用 */
        void (*set_curr_task) (struct rq *rq);
        /* 该函数通常调用自 time tick 函数；它可能引起进程切换。
        这将驱动运行时（running）抢占 */
        void (*task_tick) (struct rq *rq, struct task_struct *p, int queued);
        /* 在进程创建时调用，不同调度策略的进程初始化不一样 */
        void (*task_fork) (struct task_struct *p);
        /* 在进程退出时会使用 */
        void (*task_dead) (struct task_struct *p);

        /* 用于进程切换 */
        void (*switched_from) (struct rq *this_rq, struct task_struct *task);
        void (*switched_to) (struct rq *this_rq, struct task_struct *task);
        /* 改变优先级 */
        void (*prio_changed) (struct rq *this_rq, struct task_struct *task,
                  int oldprio);

        unsigned int (*get_rr_interval) (struct rq *rq,
```

```
                        struct task_struct *task);

        void (*update_curr) (struct rq *rq);

    #ifdef CONFIG_FAIR_GROUP_SCHED
        void (*task_move_group) (struct task_struct *p, int on_rq);
    #endif
    };
```

这个调度类具体有什么用呢，实际上在内核中不同的调度算法它们的操作都不相同，为了方便修改、替换调度算法，使用了调度类，每个调度算法只需要实现自己的调度类就可以了，CFS算法有它的调度类，SCHED_FIFO也有它自己的调度类，当一个进程创建时，用什么调度算法就将其task_struct->sched_class 指向其相应的调度类，调度器每次调度处理时，就通过当前进程的调度类函数进程操作，大大提高了可移植性和易修改性。

CFS运行队列(struct cfs_rq)

我们现在知道，在系统中至少有一个CFS运行队列，其就是根CFS运行队列，而其他的进程组和进程都包含在此运行队列中，不同的是进程组又有它自己的CFS运行队列，其运行队列中包含的是此进程组中的所有进程。当调度器从根CFS运行队列中选择了一个进程组进行调度时，进程组会从自己的CFS运行队列中选择一个调度实体进行调度(这个调度实体可能为进程，也可能又是一个子进程组)，就这样一直深入，直到最后选出一个进程进行运行为止。

对于 struct cfs_rq 结构没有什么好说明的，只要确定其代表着一个CFS运行队列，并且包含有一个红黑树进行选择调度进程即可。

```
    /* CFS调度的运行队列，每个CPU的rq会包含一个cfs_rq，而每个组调度的sched_entity也会有自己的一个cfs_rq队列 */
    /* CFS-related fields in a runqueue */
    struct cfs_rq {
     /* CFS运行队列中所有进程的总负载 */
        struct load_weight load;
       /*
         * nr_running: cfs_rq中调度实体数量
         * h_nr_running: 只对进程组有效，其下所有进程组中cfs_rq的nr_running之和
         */
        unsigned int nr_running, h_nr_running;
        /*获取当前cfs_rq的active time*/
        u64 exec_clock;
         /* 当前CFS队列上最小运行时间，单调递增
         * 两种情况下更新该值：
         * 1、更新当前运行任务的累计运行时间时
         * 2、当任务从队列删除去，如任务睡眠或退出，这时候会查看剩下的任务的vruntime是否大于min_vruntime，如果是则更新该值。
         */
        u64 min_vruntime;
```

```
#ifndef CONFIG_64BIT
    u64 min_vruntime_copy;
#endif
  /* 该红黑树的root */
    struct rb_root tasks_timeline;
     /* 下一个调度结点(红黑树最左边结点，最左边结点就是下个调度实体) */
    struct rb_node *rb_leftmost;

    /*
     * 'curr' points to currently running entity on this cfs_rq.
     * It is set to NULL otherwise (i.e when none are currently running).
     */
/*
     * curr：当前正在运行的sched_entity（对于组虽然它不会在cpu上运行，但是当它的下层有一个task在cpu上运行，那么它所在的cfs_rq就把它当做是该cfs_rq上当
前正在运行的sched_entity）
     * next：表示有些进程急需运行，即使不遵从CFS调度也必须运行它，调度时会检查是否next需要调度，有就调度next
     *
     * skip：略过进程(不会选择skip指定的进程调度)
     */

    struct sched_entity *curr, *next, *last, *skip;

#ifdef  CONFIG_SCHED_DEBUG
    unsigned int nr_spread_over;
#endif

#ifdef CONFIG_SMP
    /*
     * CFS load tracking 目的是判断哪个task进入runnable/running状态，还是需要进行balance
     */
    struct sched_avg avg;
    u64 runnable_load_sum;
    unsigned long runnable_load_avg;
#ifdef CONFIG_64BIT_ONLY_CPU
    unsigned long runnable_load_avg_32bit;
#endif
#ifdef CONFIG_FAIR_GROUP_SCHED
    unsigned long tg_load_avg_contrib;
    unsigned long propagate_avg;
#endif
    atomic_long_t removed_load_avg, removed_util_avg;
#ifndef CONFIG_64BIT
    u64 load_last_update_time_copy;
#endif

#ifdef CONFIG_FAIR_GROUP_SCHED
    /*
     *   h_load = weight * f(tg)
     *
     * Where f(tg) is the recursive weight fraction assigned to
     * this group.
```

```
     */
    unsigned long h_load;
    u64 last_h_load_update;
    struct sched_entity *h_load_next;
#endif /* CONFIG_FAIR_GROUP_SCHED */
#endif /* CONFIG_SMP */

#ifdef CONFIG_FAIR_GROUP_SCHED
   /* 所属于的CPU rq */
    struct rq *rq;  /* cpu runqueue to which this cfs_rq is attached */

    /*
     * leaf cfs_rqs are those that hold tasks (lowest schedulable entity in
     * a hierarchy). Non-leaf lrqs hold other higher schedulable entities
     * (like users, containers etc.)
     *
     * leaf_cfs_rq_list ties together list of leaf cfs_rq's in a cpu. This
     * list is used during load balance.
     */
    int on_list;
    struct list_head leaf_cfs_rq_list;
    /*属于这个cfs_rq的进程组*/
    struct task_group *tg;  /* group that "owns" this runqueue */

#ifdef CONFIG_SCHED_WALT
    /*walt 计算cfs_rq这个rq总共的runnable时间*/
    u64 cumulative_runnable_avg;
#endif
 /*cfs_rq带宽限制信息*/
#ifdef CONFIG_CFS_BANDWIDTH
    int runtime_enabled;
    u64 runtime_expires;
    s64 runtime_remaining;

    u64 throttled_clock, throttled_clock_task;
    u64 throttled_clock_task_time;
    int throttled, throttle_count, throttle_uptodate;
    struct list_head throttled_list;
#endif /* CONFIG_CFS_BANDWIDTH */
#endif /* CONFIG_FAIR_GROUP_SCHED */
};
```

- **load**：其保存的是进程组中所有进程的权值总和，需要注意子进程计算vruntime时需要用到进程组的load。

CPU运行队列(struct rq)

每个CPU都有自己的 struct rq 结构，其用于描述在此CPU上所运行的所有进程，其包括一个实时进程队列和一个根CFS运行队列，在调度时，调度器首先会先去实时进程队列找是否有实时进程需要运行，如果没有才会去CFS运行队列找是否有进行需要运行，这就是为什么常说的实时进程优先级比普通进程高，不仅仅体现在prio优先级上，还体现在调度器的设计上，至于dl运

行队列，我暂时还不知道有什么用处，其优先级比实时进程还高，但是创建进程时如果创建的是dl进程创建会错误(具体见sys_fork)。

```
/*
 * This is the main, per-CPU runqueue data structure.
 *
 * Locking rule: those places that want to lock multiple runqueues
 * (such as the load balancing or the thread migration code), lock
 * acquire operations must be ordered by ascending &runqueue.
 */
struct rq {
    /* runqueue lock: */
    raw_spinlock_t lock;

    /*
     * nr_running and cpu_load should be in the same cacheline because
     * remote CPUs use both these fields when doing load calculation.
     */
    /*这个rq里面存在多少个running task，包括RT，fair，DL sched class的task*/
    unsigned int nr_running;
#ifdef CONFIG_NUMA_BALANCING
    unsigned int nr_numa_running;
    unsigned int nr_preferred_running;
#endif
    #define CPU_LOAD_IDX_MAX 5
    unsigned long cpu_load[CPU_LOAD_IDX_MAX];
    unsigned long last_load_update_tick;
    /*在选择下一个调度实体的时候，需要判断此task是否是misfit task，是否做的决策是
     不相同的，比如会强制balance等等*/
    unsigned int misfit_task;
#ifdef CONFIG_NO_HZ_COMMON
    u64 nohz_stamp;
    unsigned long nohz_flags;
#endif
#ifdef CONFIG_NO_HZ_FULL
    unsigned long last_sched_tick;
#endif

#ifdef CONFIG_CPU_QUIET
    /* time-based average load */
    u64 nr_last_stamp;
    u64 nr_running_integral;
    seqcount_t ave_seqcnt;
#endif

    /* capture load from *all* tasks on this cpu: */
    /*在rq里面的可运行的所有task的总的load，当nr_running数量发生变化时也会更新*/
    struct load_weight load;
    /*在rq里面有多少个task的load需要更新*/
    unsigned long nr_load_updates;
    /*进程发生上下文切换的次数，只有proc 文件系统里面会导出这个统计数值*/
    u64 nr_switches;
    /*每个cpu上的rq，都包含了cfs_rq,rt_rq和dl_rq调度队列，包含红黑树的根*/
```

```
        struct cfs_rq cfs;
        struct rt_rq rt;
        struct dl_rq dl;

    #ifdef CONFIG_FAIR_GROUP_SCHED
        /* list of leaf cfs_rq on this cpu: */
        struct list_head leaf_cfs_rq_list;
        struct list_head *tmp_alone_branch;
    #endif /* CONFIG_FAIR_GROUP_SCHED */

        /*
         * This is part of a global counter where only the total sum
         * over all CPUs matters. A task can increase this counter on
         * one CPU and if it got migrated afterwards it may decrease
         * it on another CPU. Always updated under the runqueue lock:
         *//* 曾经处于队列但现在处于TASK_UNINTERRUPTIBLE状态的进程数量 */
        unsigned long nr_uninterruptible;
        /*curr指针表示当前运行的task指针，idle表示rq里面没有其他进程可以运行，最后执行
        idle task，此cpu进入idle状态，stop表示当前task sched_class stop调度类*/
        struct task_struct *curr, *idle, *stop;
        /*下一次balance的时间，系统会周期性的做balance动作。*/
        unsigned long next_balance;
        struct mm_struct *prev_mm;

        unsigned int clock_skip_update;
        /*rq运行时间，是一个累加值*/
        u64 clock;
        u64 clock_task;
        /*当前rq里面有多少iowait数量*/
        atomic_t nr_iowait;

    #ifdef CONFIG_SMP
        struct root_domain *rd;
        /* 当前CPU所在基本调度域，每个调度域包含一个或多个CPU组，每个CPU组包含该调度
        域中一个或多个CPU子集，负载均衡都是在调度域中的组之间完成的，不能跨域进行负载均衡 */
        struct sched_domain *sd;
        /*cpu_capacity：此cpu的实际capacity，会随着系统运行变化而变化，初始值
         为capacity_orig，cpu_capacity_orig：是dts配置的各个cpu的capacity，是一个常量
    */
        unsigned long cpu_capacity;
        unsigned long cpu_capacity_orig;

        struct callback_head *balance_callback;

        unsigned char idle_balance;
        /* For active balancing */
        /* 如果需要把进程迁移到其他运行队列，就需要设置这个位 */
        int active_balance;
        int push_cpu;
        struct task_struct *push_task;
        struct cpu_stop_work active_balance_work;
        /* cpu of this runqueue: */
        /* 该运行队列所属CPU */
        int cpu;
```

```
        int online;

        struct list_head cfs_tasks;

    #ifdef CONFIG_INTEL_DWS
        struct intel_dws dws;
    #endif
        /*rt task的负载，随着sched period周期衰减一半。看这个函数：sched_avg_update*/
        u64 rt_avg;
        /* 该运行队列存活时间,区别于rq运行时间，数值update在一个cpu启动的时候和调度器
        初始化的时候*/
        u64 age_stamp;
        /*在某个cpu变成idle的时候，标记rq idle的时间戳*/
        u64 idle_stamp;
        /*rq平均idle的时间*/
        u64 avg_idle;

        /* This is used to determine avg_idle's max value */
        u64 max_idle_balance_cost;
    #endif
    /*在WALT window assist load tracing的文章中详细的讲解了这几个参数怎么计算，怎么
    实现的*/
    #ifdef CONFIG_SCHED_WALT
        u64 cumulative_runnable_avg;
        u64 window_start;
        u64 curr_runnable_sum;
        u64 prev_runnable_sum;
        u64 nt_curr_runnable_sum;
        u64 nt_prev_runnable_sum;
        u64 cur_irqload;
        u64 avg_irqload;
        u64 irqload_ts;
        u64 cum_window_demand;
        /*为了解决某一个问题，我们自己添加的几个flag，目的是针对高负载（两个窗口都是高负载）
        情况下使用前一个窗口此task的running time来计算此task在当前窗口的util
        数值（cpu_util_freq此函数计算）。
        现在的系统使用的是此task在当前窗口的比重来计算util的，可能会存在频率不稳定的情况*/
        enum {
            CPU_BUSY_CLR = 0,
            CPU_BUSY_PREPARE,
            CPU_BUSY_SET,
        } is_busy;
    #endif /* CONFIG_SCHED_WALT */


    #ifdef CONFIG_IRQ_TIME_ACCOUNTING
        /*计算irq起来的时间戳*/
        u64 prev_irq_time;
    #endif
    #ifdef CONFIG_PARAVIRT
        u64 prev_steal_time;
    #endif
    #ifdef CONFIG_PARAVIRT_TIME_ACCOUNTING
        u64 prev_steal_time_rq;
```

```
#endif

    /* calc_load related fields */
    /*负载均衡相关*/
    unsigned long calc_load_update;
    long calc_load_active;

#ifdef CONFIG_SCHED_HRTICK
#ifdef CONFIG_SMP
    int hrtick_csd_pending;
    struct call_single_data hrtick_csd;
#endif
    struct hrtimer hrtick_timer;
#endif
/*统计调度信息使用*/
#ifdef CONFIG_SCHEDSTATS
    /* latency stats */
    struct sched_info rq_sched_info;
    unsigned long long rq_cpu_time;
    /* could above be rq->cfs_rq.exec_clock + rq->rt_rq.rt_runtime ? */

    /* sys_sched_yield() stats */
    unsigned int yld_count;

    /* schedule() stats */
    unsigned int sched_count;
    unsigned int sched_goidle;

    /* try_to_wake_up() stats */
    unsigned int ttwu_count;
    unsigned int ttwu_local;
#ifdef CONFIG_SMP
    /*统计eas状态信息，由于看的arm新的调度器是基于EAS实现的*/
    struct eas_stats eas_stats;
#endif
#endif

#ifdef CONFIG_SMP
    struct llist_head wake_list;
#endif

#ifdef CONFIG_CPU_IDLE
    /* Must be inspected within a rcu lock section */
    /*在cpuidle_enter_state中设置，即cpu进入相应的idle state时候才会设置这
    两个参数*/
    struct cpuidle_state *idle_state;
    int idle_state_idx;
#endif
};
```

0号进程初始化调度器相关结构体，并将0号进程调度类变换为idle_sched_class调度类

从arch/arm64/kernel/head.S第一次启动从汇编代码执行C语言代码的函数为start_kernel：

start_kernel----->sched_init()，下面看看sched_init函数的实现过程，我们对照函数来讲解：

```
void __init sched_init(void)
{
    int i, j;
    unsigned long alloc_size = 0, ptr;

#ifdef CONFIG_FAIR_GROUP_SCHED
    /*为cfs_rq,sched_entity分配空间，每个cpu都存在cfs_rq和se*/
    alloc_size += 2 * nr_cpu_ids * sizeof(void **);
#endif
#ifdef CONFIG_RT_GROUP_SCHED
   /*为rt_rq和rt_sched_entity分配空间，每个cpu都存在*/
    alloc_size += 2 * nr_cpu_ids * sizeof(void **);
#endif
    if (alloc_size) {
        ptr = (unsigned long)kzalloc(alloc_size, GFP_NOWAIT);

#ifdef CONFIG_FAIR_GROUP_SCHED
        /*root_task_group作为RB tree的根节点，作为所有task的
       跟节点(不管是单个task还是进程组task) 根据上面alloc_size地址空间大小和
        ptr作为alloc_size这块空间大小的首地址进行分配
        可以看到分别每个cpu上的cfs_rq和se分配空间，*/
        root_task_group.se = (struct sched_entity **)ptr;
        ptr += nr_cpu_ids * sizeof(void **);

        root_task_group.cfs_rq = (struct cfs_rq **)ptr;
        ptr += nr_cpu_ids * sizeof(void **);

#endif /* CONFIG_FAIR_GROUP_SCHED */
#ifdef CONFIG_RT_GROUP_SCHED
        /*同上，为rt相关分配空间，也就是说root_task_group包含了cfs task
         rt task*/
        root_task_group.rt_se = (struct sched_rt_entity **)ptr;
        ptr += nr_cpu_ids * sizeof(void **);

        root_task_group.rt_rq = (struct rt_rq **)ptr;
        ptr += nr_cpu_ids * sizeof(void **);

#endif /* CONFIG_RT_GROUP_SCHED */
    }
#ifdef CONFIG_CPUMASK_OFFSTACK
    for_each_possible_cpu(i) {
        per_cpu(load_balance_mask, i) = (cpumask_var_t)kzalloc_node(
            cpumask_size(), GFP_KERNEL, cpu_to_node(i));
    }
#endif /* CONFIG_CPUMASK_OFFSTACK */
    /*初始化rt thread的带宽限制，以1s一个周期计算，如果rt在rt_rq里面运行时间超过
   950ms则强制将task出rt_rq队列，等到下一个1s周期再次运行，每次循环往复知道thread
    运行完毕*/
    init_rt_bandwidth(&def_rt_bandwidth,
            global_rt_period(), global_rt_runtime());
    /*意思同上，但是dl thread不知道是什么thread？？？？*/
    init_dl_bandwidth(&def_dl_bandwidth,
```

```
            global_rt_period(), global_rt_runtime());

#ifdef CONFIG_SMP
    /*更新root domain，并初始化max_cpu_capacity结构体，update_cpu_capacity时候
    会被使用到*/
    init_defrootdomain();
#endif

#ifdef CONFIG_RT_GROUP_SCHED
    /*为root_task_group内的rt task设置带宽限制*/
    init_rt_bandwidth(&root_task_group.rt_bandwidth,
            global_rt_period(), global_rt_runtime());
#endif /* CONFIG_RT_GROUP_SCHED */

#ifdef CONFIG_CGROUP_SCHED
    /*将root_task_group加入到task group链表中，并设置它的孩子节点和兄弟节点链表*/
    list_add(&root_task_group.list, &task_groups);
    INIT_LIST_HEAD(&root_task_group.children);
    INIT_LIST_HEAD(&root_task_group.siblings);
    autogroup_init(&init_task);

#endif /* CONFIG_CGROUP_SCHED */
    /*每个cpu都有一个rq，开始为所有cpu上的rq进行初始化*/
    for_each_possible_cpu(i) {
        struct rq *rq;

        rq = cpu_rq(i); /*使用per_cpu关联cpu与rq*/
        raw_spin_lock_init(&rq->lock);
        /*系统初始化时，rq队列里面没有可运行的task，置为0*/
        rq->nr_running = 0;
        rq->calc_load_active = 0;
        /*计算负载update period*/
        rq->calc_load_update = jiffies + LOAD_FREQ;
        /*初始化rq里面的cfs_rq,rt_rq和dl_rq
        1. 初始化cfs_rq rb tree的root node和vruntime，这是主要的挑选哪个rb
        node运行的关键，以后在分析，主要是一些cfs_rq成员变量的初始化*/
        init_cfs_rq(&rq->cfs);
        init_rt_rq(&rq->rt);
        init_dl_rq(&rq->dl);
#ifdef CONFIG_FAIR_GROUP_SCHED
        root_task_group.shares = ROOT_TASK_GROUP_LOAD;
        INIT_LIST_HEAD(&rq->leaf_cfs_rq_list);
        rq->tmp_alone_branch = &rq->leaf_cfs_rq_list;
        /*
         * How much cpu bandwidth does root_task_group get?
         *
         * In case of task-groups formed thr' the cgroup filesystem, it
         * gets 100% of the cpu resources in the system. This overall
         * system cpu resource is divided among the tasks of
         * root_task_group and its child task-groups in a fair manner,
         * based on each entity's (task or task-group's) weight
         * (se->load.weight).
         *
         * In other words, if root_task_group has 10 tasks of weight
```

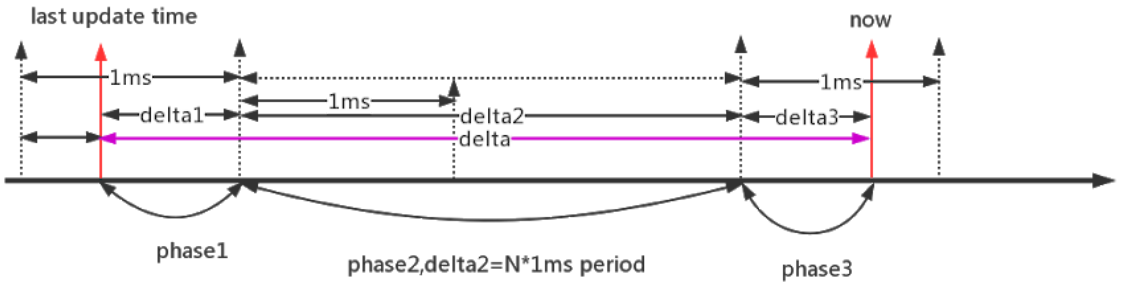
```
             * 1024) and two child groups A0 and A1 (of weight 1024 each),
             * then A0's share of the cpu resource is:
             *
             *  A0's bandwidth = 1024 / (10*1024 + 1024 + 1024) = 8.33%
             *
             * We achieve this by letting root_task_group's tasks sit
             * directly in rq->cfs (i.e root_task_group->se[] = NULL).
             */
            /*初始化cfs task的带宽限制，比rt要复杂些，以后在展开讲解*/
            init_cfs_bandwidth(&root_task_group.cfs_bandwidth);
            /*上面分析task_group结构体的时候，我们知道一个进程组里面的进程可能处在不同
            的CPU上运行，那么也就是不同的cfs_rq和调度实体了，所以每个task都会标记是属于
             哪个cfs_rq和调度实体的。*/
            init_tg_cfs_entry(&root_task_group, &rq->cfs, NULL, i, NULL);
    #endif /* CONFIG_FAIR_GROUP_SCHED */
             /*初始化rt可以运行的时间，默认值为950ms*/
            rq->rt.rt_runtime = def_rt_bandwidth.rt_runtime;
    #ifdef CONFIG_RT_GROUP_SCHED
            /*初始化rt的rq和se*/
            init_tg_rt_entry(&root_task_group, &rq->rt, NULL, i, NULL);
    #endif

            for (j = 0; j < CPU_LOAD_IDX_MAX; j++)
                rq->cpu_load[j] = 0;
            /*标记load last update time*/
            rq->last_load_update_tick = jiffies;

    #ifdef CONFIG_SMP
            rq->sd = NULL;
            rq->rd = NULL;
            /*初始化rq的capacity数值，来自dts，但是rq->cpu_capacity数值会随着系统变化
            ，这个有待解惑，目前我也没找到root cause*/
            rq->cpu_capacity = rq->cpu_capacity_orig = SCHED_CAPACITY_SCALE;
            /*下面三个与balance相关，active_balance是一个flag，目的在于实现负载的强制
            balance，而next_balance是一个周期性的tick，记录周期性balance的时间*/
            rq->balance_callback = NULL;
            rq->active_balance = 0;
            rq->next_balance = jiffies;
            rq->push_cpu = 0;
            rq->push_task = NULL;
            /*rq所处的cpu*/
            rq->cpu = i;
            rq->online = 0;
            /*rq idle，也就是cpu idle的时间戳*/
            rq->idle_stamp = 0;
            rq->avg_idle = 2*sysctl_sched_migration_cost;
            rq->max_idle_balance_cost = sysctl_sched_migration_cost;
    #ifdef CONFIG_SCHED_WALT
            /*在WALT中会计算处理每个irq的time，并转化为load*/
            rq->cur_irqload = 0;/*当前窗口的irq运行时间*/
            rq->avg_irqload = 0; /*irq可能跨多个窗口运行，经过衰减算法，计算多
            的irq运行时间，作为avg_irqload个窗口*/
            /*irq enter/exit时间戳*/
            rq->irqload_ts = 0;
```

```
        /*我们自己新添加的一个flag，目的为了性能*/
        rq->is_busy = CPU_BUSY_CLR;
#endif
        /*初始化cfs tasks的链表头*/
        INIT_LIST_HEAD(&rq->cfs_tasks);
        /*将rq挂载在默认的root domain上，对于domain还需要仔细的check！！！*/
        rq_attach_root(rq, &def_root_domain);
#ifdef CONFIG_NO_HZ_COMMON
        rq->nohz_flags = 0;
#endif
#ifdef CONFIG_NO_HZ_FULL
        rq->last_sched_tick = 0;
#endif
#endif  /*初始化rq的hrtimer*/
        init_rq_hrtick(rq);
        /*设置rq iowait数值为0*/
        atomic_set(&rq->nr_iowait, 0);

#ifdef CONFIG_INTEL_DWS
        init_intel_dws(rq);
#endif
    }  /*至此，整个rq初始化完毕*/
    /*设置init_task load权重,每个task会根据task的优先级分配不同的权重值*/
    set_load_weight(&init_task);

#ifdef CONFIG_PREEMPT_NOTIFIERS
    /*初始化抢占通知链*/
    INIT_HLIST_HEAD(&init_task.preempt_notifiers);
#endif

    /*
     * The boot idle thread does lazy MMU switching as well:
     */
    atomic_inc(&init_mm.mm_count);
    enter_lazy_tlb(&init_mm, current);

    /*
     * During early bootup we pretend to be a normal task:
     */
    /*设置当前task为fair调度类，current也就是init_task thread*/
    current->sched_class = &fair_sched_class;

    /*
     * Make us the idle thread. Technically, schedule() should not be
     * called from this thread, however somewhere below it might be,
     * but because we are the idle thread, we just pick up running again
     * when this runqueue becomes "idle".
     *//*将当前进程初始化为idle进程
      比较有意思，但是最重要的一点就是，设置它的调度类为idle_sched_class*/
    init_idle(current, smp_processor_id());
    /*下次load update的时间*/
    calc_load_update = jiffies + LOAD_FREQ;

#ifdef CONFIG_SMP
```

```
        zalloc_cpumask_var(&sched_domains_tmpmask, GFP_NOWAIT);
        /* May be allocated at isolcpus cmdline parse time */
        if (cpu_isolated_map == NULL)
            zalloc_cpumask_var(&cpu_isolated_map, GFP_NOWAIT);
        /*将boot cpu上当前task设置为idle_thread，对于其他从cpu则在idle_threads_init里
         面，为每个cpu fork出idle_threads*/
        idle_thread_set_boot_cpu();
        /*设置rq age_stamp，即rq的启动时间(包括idle和running时间)，不是运行时间，*/
        set_cpu_rq_start_time();
    #endif
        init_sched_fair_class();

    #ifdef CONFIG_64BIT_ONLY_CPU
        arch_get_64bit_only_cpus(&b64_only_cpu_mask);
    #ifdef CONFIG_SCHED_COMPAT_LIMIT
        /* get cpus that support AArch32 and store in compat_32bit_cpu_mask */
        cpumask_andnot(&compat_32bit_cpu_mask, cpu_present_mask,
            &b64_only_cpu_mask);
    #endif
    #endif
        /*调度器开始工作了*/
        scheduler_running = 1;
    }
```

- 新创建的进程
- idle进程被wakeup

它们是怎样加入调度rq的




```
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;
        delta_w = 1024 - delta_w;
       /*phase1 start*/
        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;
       /*phase1 end*/
        delta -= delta_w;
        /* Figure out how many additional periods this update spans */
        periods = delta / 1024;
        delta %= 1024;
        /*phase2 start*/
        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;
    }
    /*phase2 end*/
   /*phase3 start*/
    /* 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;
   /*phase3 end*/
    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;
    }
```

phase1,delta1

phase2,delta2

phase3,delta3

注意root_task_group

```
/*
 * Default task group.
 * Every task in system belongs to this group at bootup.
 */
struct task_group root_task_group;
```

在sched_init函数中怎么为每一个cpu的cfs_rq，se初始化的

1. 怎么计算load weight：

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

    /*
     * SCHED_IDLE tasks get minimal weight:
     */
    if (idle_policy(p->policy)) {
        load->weight = scale_load(WEIGHT_IDLEPRIO);
        load->inv_weight = WMULT_IDLEPRIO;
        return;
    }

    load->weight = scale_load(prio_to_weight[prio]);
    load->inv_weight = prio_to_wmult[prio];
}
#define WEIGHT_IDLEPRIO                3
#define WMULT_IDLEPRIO         1431655765

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

/*
 * 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,
};
```