（六）Linux进程调度-实时调度器

原创 LoyenWang LoyenWang 3月27日

来自专辑

Linux进程管理

背景

- `Read the fucking source code!` --By 鲁迅
- `A picture is worth a thousand words.` --By 高尔基

说明：

1. Kernel版本：4.14
2. ARM64处理器，Contex-A53，双核
3. 使用工具：Source Insight 3.5， Visio

1. 概述

在Linux内核中，实时进程总是比普通进程的优先级要高，实时进程的调度是由 `Real Time Scheduler(RT调度器)` 来管理，而普通进程由 `CFS调度器` 来管理。实时进程支持的调度策略为： `SCHED_FIFO` 和 `SCHED_RR` 。

前边的系列文章都是针对 `CFS调度器` 来分析的，包括了 `CPU负载` 、 `组调度` 、 `Bandwidth控制` 等，本文的 `RT调度器` 也会从这些角度来分析，如果看过之前的系列文章，那么这篇文章理解起来就会更容易点了。

前戏不多，直奔主题。

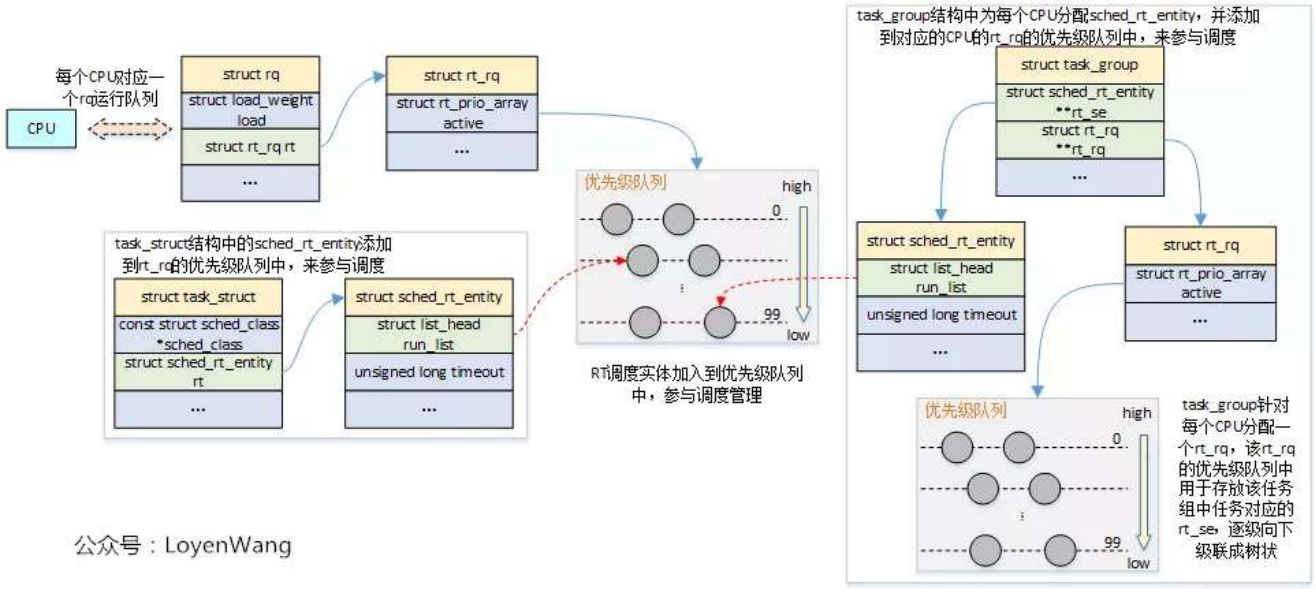
2. 数据结构

有必要把关键的结构体罗列一下了：

- `struct rq` ：运行队列，每个CPU都对应一个；
- `struct rt_rq` ：实时运行队列，用于管理实时任务的调度实体；

- `struct sched_rt_entity` ：实时调度实体，用于参与调度，功能与 `struct sched_entity` 类似；
- `struct task_group` ：组调度结构体；
- `struct rt_bandwidth` ：带宽控制结构体；

老规矩，先上张图，捋捋这些结构之间的关系吧：



- 从图中的结构组织关系看，与 CFS调度器 基本一致，区别在与 CFS调度器 是通过红黑树来组织调度实体，而 RT调度器 使用的是优先级队列来组织实时调度实体；
- `rt_rq` 运行队列，维护了100个优先级的队列（链表），优先级0-99，从高到底；
- 调度器管理的对象是调度实体，任务 `task_struct` 和任务组 `task_group` 都是通过内嵌调度实体的数据结构，来最终参与调度管理的；
- `task_group` 任务组调度，自身为每个CPU维护 `rt_rq` ，用于存放自己的子任务或者子任务组，子任务组又能往下级联，因此可以构造成树；

上述结构体中， `struct rq` 和 `struct task_group` ，在前文中都分析过。下边针对RT运行队列相关的关键结构体，简单注释下吧：

```
struct sched_rt_entity {
        struct list_head        run_list;    //用于加入到优先级队列中
        unsigned long           timeout;     //设置的时间超时
        unsigned long           watchdog_stamp;      //用于记录jiffies值
        unsigned int            time_slice;      //时间片，100ms，
        unsigned short          on_rq;
        unsigned short          on_list;

```

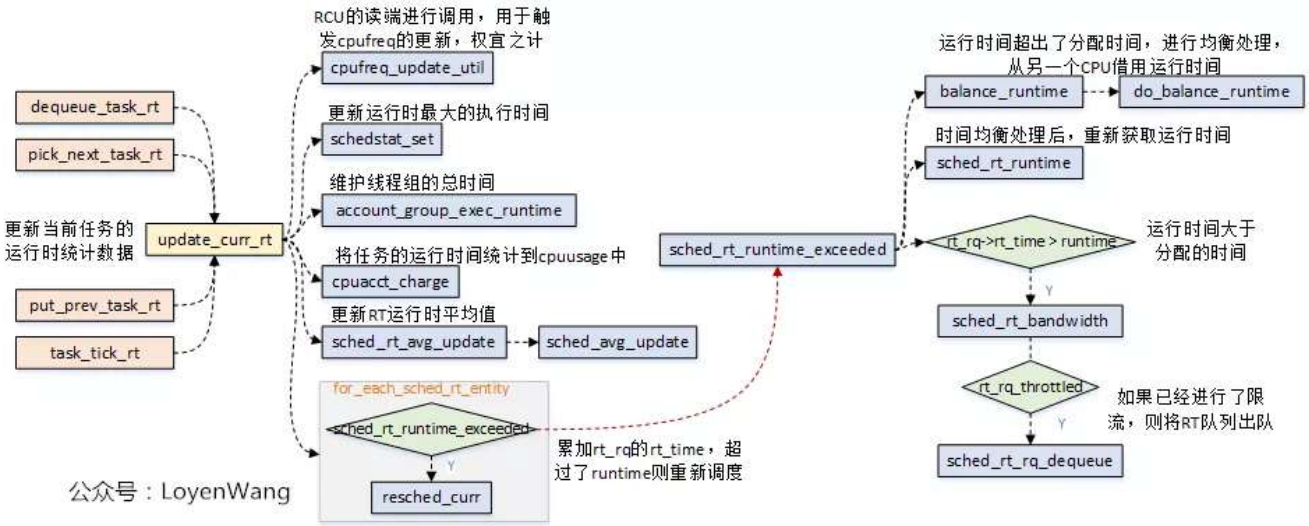
```
        struct sched_rt_entity          *back;  //临时用于从上往下连接RT调度实体时
#ifdef CONFIG_RT_GROUP_SCHED
        struct sched_rt_entity          *parent;    //指向父RT调度实体
        /* rq on which this entity is (to be) queued: */
        struct rt_rq                    *rt_rq;     //RT调度实体所属的实时运行队列
        /* rq "owned" by this entity/group: */
        struct rt_rq                    *my_q;  //RT调度实体所拥有的实时运行队列，
#endif
} __randomize_layout;


/* Real-Time classes' related field in a runqueue: */
struct rt_rq {
        struct rt_prio_array active;    //优先级队列，100个优先级的链表，并定义了位
        unsigned int rt_nr_running; //在RT运行队列中所有活动的任务数
        unsigned int rr_nr_running;
#if defined CONFIG_SMP || defined CONFIG_RT_GROUP_SCHED
        struct {
                int curr; /* highest queued rt task prio */     //当前RT任务的最
#ifdef CONFIG_SMP
                int next; /* next highest */        //下一个要运行的RT任务的优先级
#endif
        } highest_prio;
#endif
#ifdef CONFIG_SMP
        unsigned long rt_nr_migratory;      //任务没有绑定在某个CPU上时，这个值会增
        unsigned long rt_nr_total;      //用于overload检查
        int overloaded;     //RT运行队列过载，则将任务推送到其他CPU
        struct plist_head pushable_tasks;   //优先级列表，用于推送过载任务
#endif /* CONFIG_SMP */
        int rt_queued;  //表示RT运行队列已经加入rq队列

        int rt_throttled;   //用于限流操作
        u64 rt_time;    //累加的运行时，超出了本地rt_runtime时，则进行限制
        u64 rt_runtime;     //分配给本地池的运行时
        /* Nests inside the rq lock: */
        raw_spinlock_t rt_runtime_lock;

#ifdef CONFIG_RT_GROUP_SCHED
        unsigned long rt_nr_boosted;    //用于优先级翻转问题解决
```

```
49
50          struct rq *rq;      //指向运行队列
51          struct task_group *tg;  //指向任务组
52  #endif
53  };
54
55  struct rt_bandwidth {
56          /* nests inside the rq lock: */
57          raw_spinlock_t          rt_runtime_lock;
58          ktime_t                 rt_period;      //时间周期
59          u64                     rt_runtime;     //一个时间周期内的运行时间，超过则
60          struct hrtimer          rt_period_timer;    //时间周期定时器
61          unsigned int            rt_period_active;
62  };
```

3. 流程分析

3.1 运行时统计数据

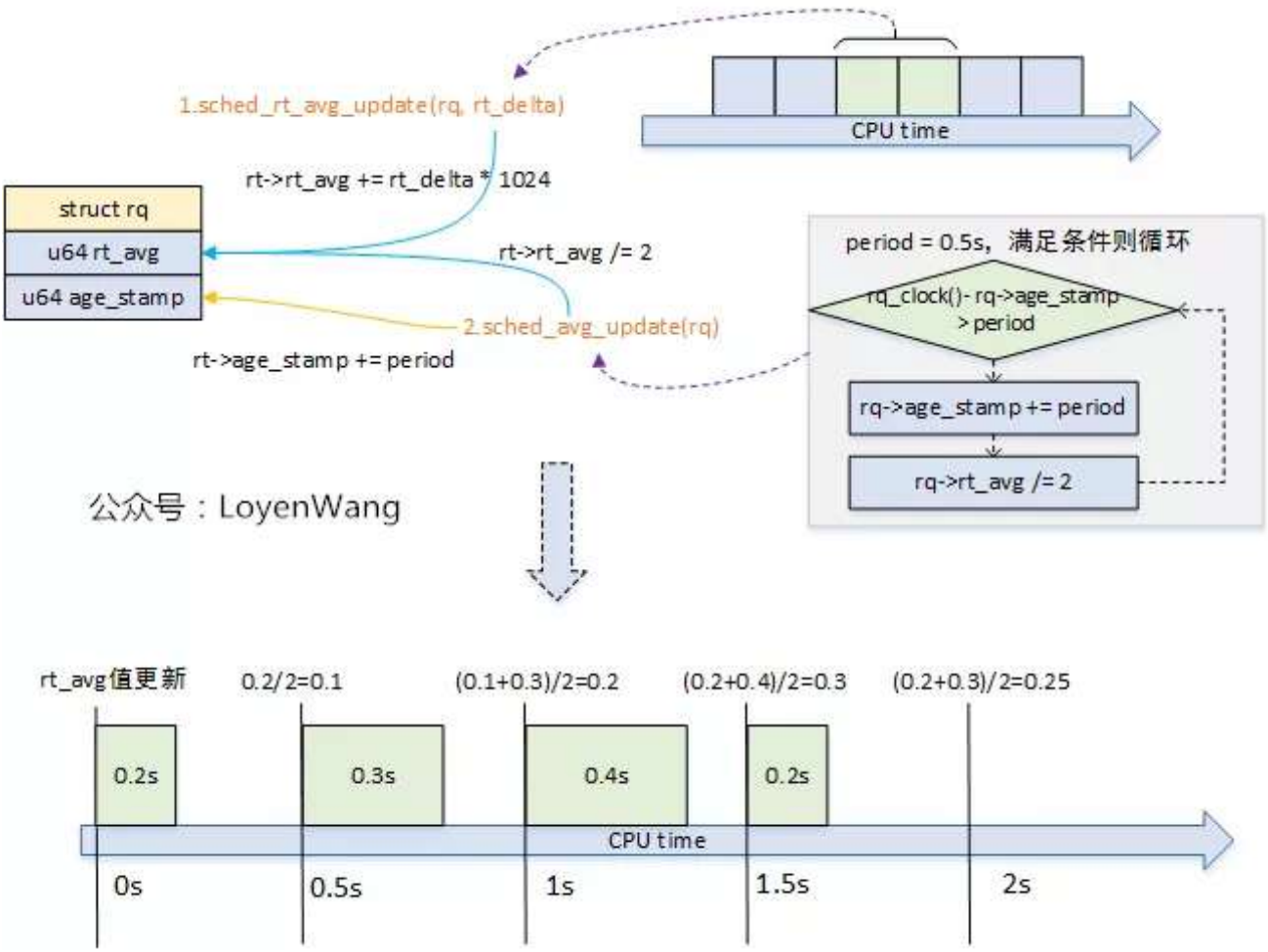
运行时的统计数据更新，是在 `update_curr_rt` 函数中完成的：



- `update_curr_rt` 函数功能，主要包括两部分：
  1. 运行时间的统计更新处理；
  2. 如果运行时间超出了分配时间，进行时间均衡处理，并且判断是否需要进行限流，进行了限流则需要将RT队列出队，并重新进行调度；

为了更直观的理解，下边还是来两张图片说明一下：

`sched_rt_avg_update` 更新示意如下：

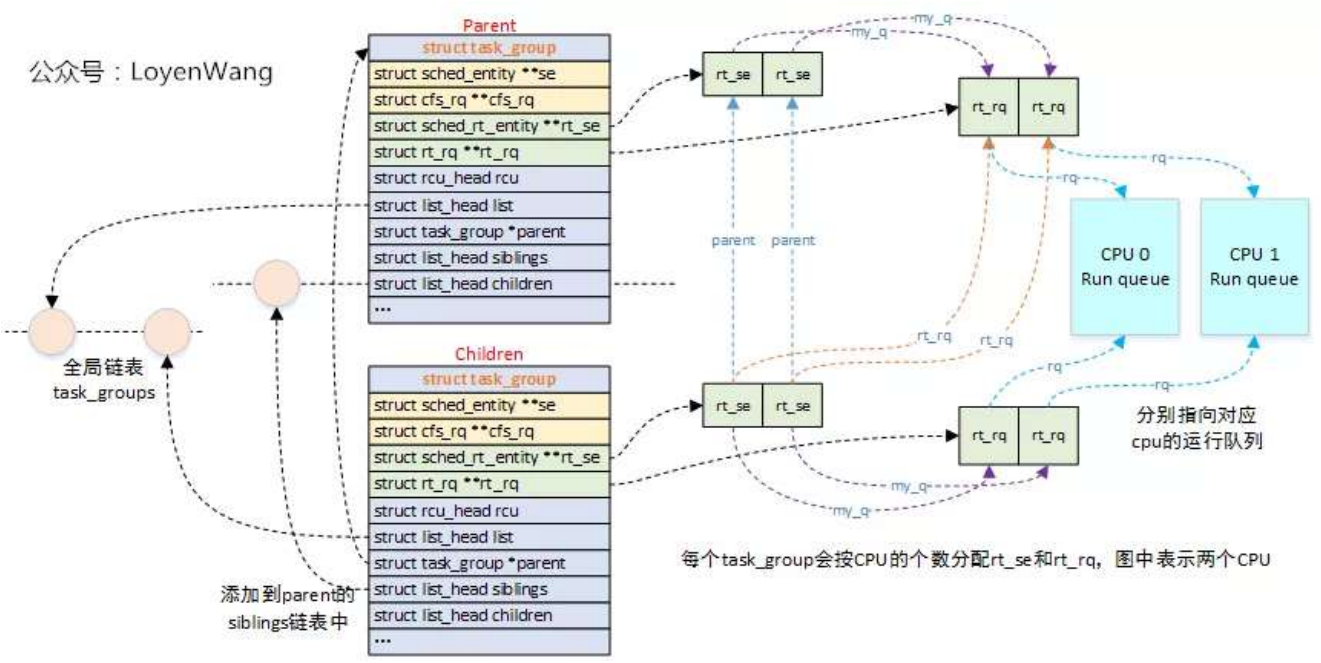


- `rq->age_stamp` ：在CPU启动后运行队列首次运行时设置起始时间，后续周期性进行更新；
- `rt_avg` ：累计的RT平均运行时间，每0.5秒减半处理，用于计算CFS负载减去RT在CFS负载平衡中使用的时间百分比；

3.2 组调度

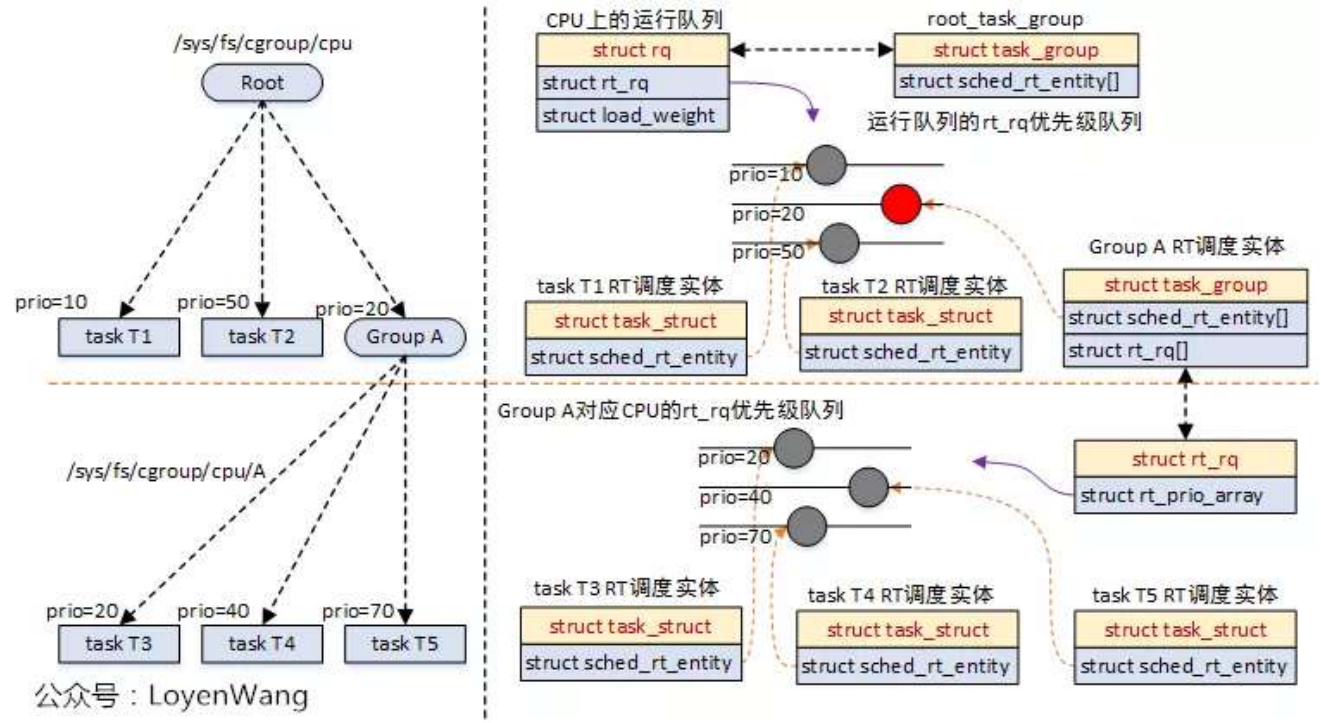
`RT调度器` 与 `CFS调度器` 的组调度基本类似， `CFS调度器` 的组调度请参考 （四）Linux进程调度-组调度及带宽控制 。

看一下 `RT调度器` 组调度的组织关系图吧：



- 系统为每个CPU都分配了RT运行队列，以及RT调度实体，任务组通过它包含的RT调度实体来参与调度；
- 任务组 `task_group` 的RT队列，用于存放归属于该组的任务或子任务组，从而形成级联的结构；

看一下实际的组织示意图：



3.3 带宽控制

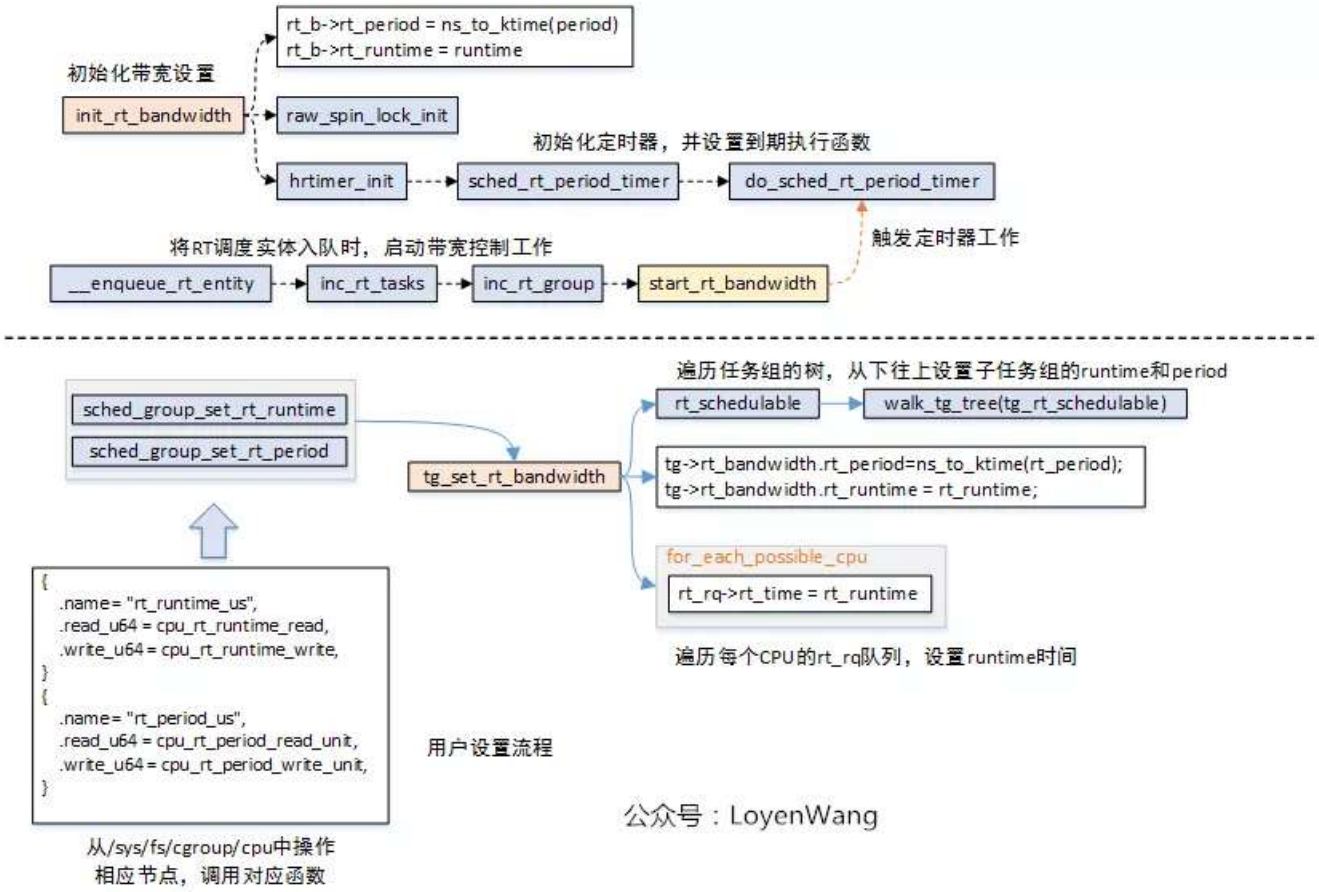
请先参考 （四）Linux进程调度-组调度及带宽控制 。

RT调度器 在带宽控制中，调度时间周期设置的为1s，运行时间设置为0.95s：

```
/*
 * period over which we measure -rt task CPU usage in us.
 * default: 1s
 */
unsigned int sysctl_sched_rt_period = 1000000;

/*
 * part of the period that we allow rt tasks to run in us.
 * default: 0.95s
 */
int sysctl_sched_rt_runtime = 950000;
```

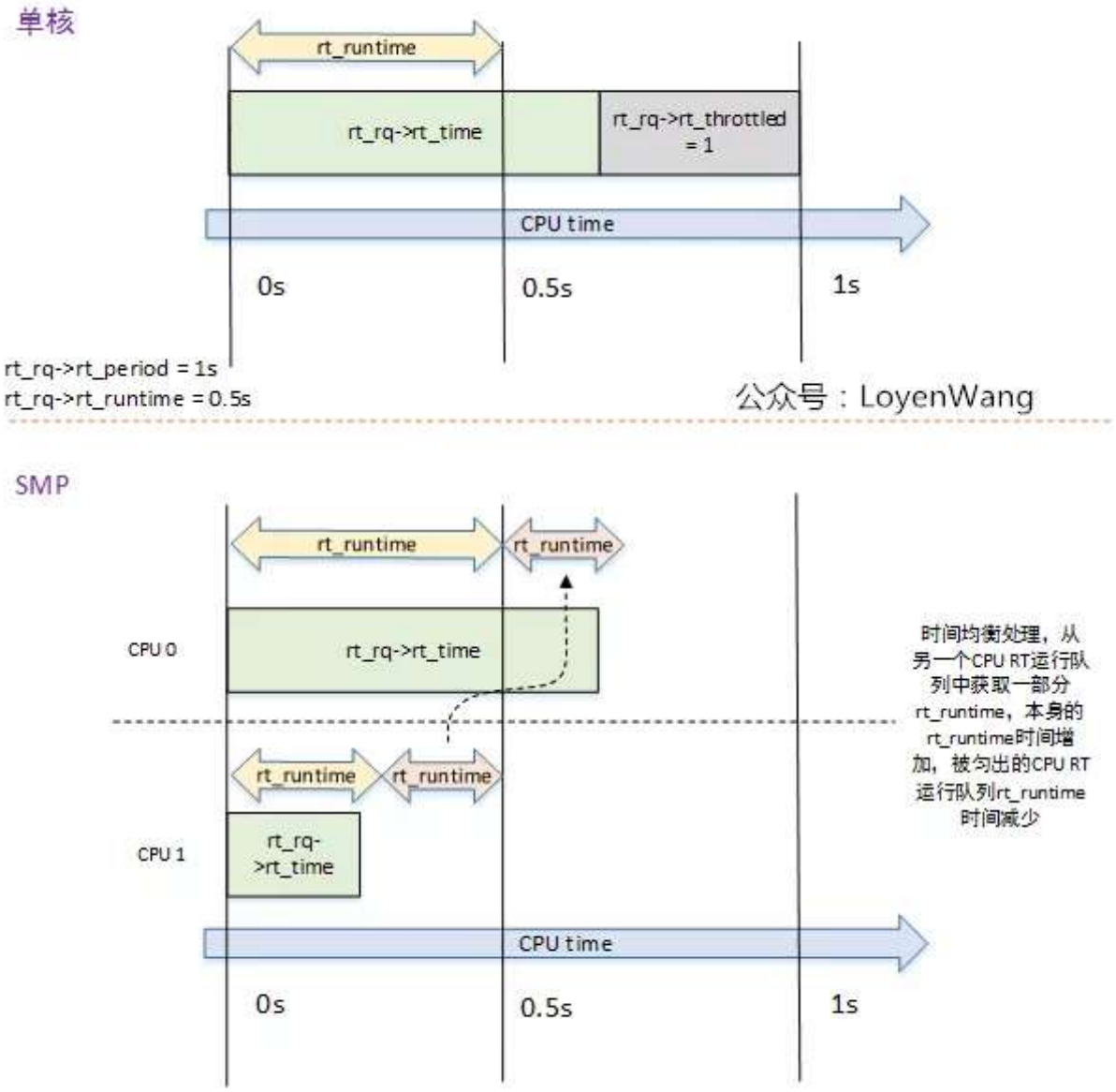
这两个值可以在用户态通过 /sys/fs/cgroup/cpu/rt_runtime_us 和 /sys/fs/cgroup/cpu/rt_period_us 来进行设置。

看看函数调用流程：



- `init_rt_bandwidth` 函数在创建分配RT任务组的时候调用，完成的工作是将 `rt_bandwidth` 结构体的相关字段进行初始化：设置好时间周期 `rt_period` 和运行时间限制 `rt_runtime` ，都设置成默认值；
- 可以从用户态通过操作 `/sys/fs/cgroup/cpu` 下对应的节点进行设置 `rt_period` 和 `rt_runtime` ，最终调用的函数是 `tg_set_rt_bandwidth` ，在该函数中会从下往上的遍历任务组进行设置时间周期和限制的运行时间；
- 在 `enqueue_rt_entity` 将RT调度实体入列时，最终触发 `start_rt_bandwidth` 函数执行，当高精度定时器到期时调用 `do_sched_rt_period_timer` 函数；
- `do_sched_rt_period_timer` 函数，会去判断该RT运行队列的累计运行时间 `rt_time` 与设置的限制运行时间 `rt_runtime` 之间的大小关系，以确定是否限流的操作。在这个函数中，如果已经进行了限流操作，会调用 `balance_time` 来在多个CPU之间进行时间均衡处理，简单点说，就是从其他CPU的rt_rq队列中匀出一部分时间增加到当前CPU的rt_rq队列中，也就是将当前rt_rt运行队列的限制运行时间 `rt_runtime` 增加一部分，其他CPU的rt_rq运行队列限制运行时间减少一部分。

来一张效果示意图：



3.4 调度器函数分析

来一张前文的图：

看一下RT调度器实例的代码：

```
const struct sched_class rt_sched_class = {
        .next                   = &fair_sched_class,
        .enqueue_task           = enqueue_task_rt,
        .dequeue_task           = dequeue_task_rt,
        .yield_task             = yield_task_rt,

        .check_preempt_curr     = check_preempt_curr_rt,

        .pick_next_task         = pick_next_task_rt,
        .put_prev_task          = put_prev_task_rt,

#ifdef CONFIG_SMP
        .select_task_rq         = select_task_rq_rt,

        .set_cpus_allowed       = set_cpus_allowed_common,
        .rq_online              = rq_online_rt,
        .rq_offline             = rq_offline_rt,
```

```
18          .task_woken             = task_woken_rt,
19          .switched_from          = switched_from_rt,
20  #endif
21
22          .set_curr_task          = set_curr_task_rt,
23          .task_tick              = task_tick_rt,
24
25          .get_rr_interval        = get_rr_interval_rt,
26
27          .prio_changed           = prio_changed_rt,
28          .switched_to            = switched_to_rt,
29
30          .update_curr            = update_curr_rt,
31  };
```

3.4.1 pick_next_task_rt

`pick_next_task_rt` 函数是调度器用于选择下一个执行任务。流程如下：

- 与 `CFS调度器` 不同， `RT调度器` 会在多个CPU组成的 `domain` 中，对任务进行 `pull/push` 处理，也就是说，如果当前CPU的运行队列中任务优先级都不高，那么会考虑去其他CPU运行队列中找一个更高优先级的任务来执行，以确保按照优先级处理，此外当前CPU也会把任务推送到其他更低优先级的CPU运行队列上。

- `_pick_next_task_rt` 的处理逻辑比较简单，如果实时调度实体是 `task` ，则直接查找优先级队列的位图中，找到优先级最高的任务，而如果实时调度实体是 `task_group` ，则还需要继续往下进行遍历查找；

关于任务的 `pull/push` ，linux提供了 `struct plist` ，基于优先级的双链表，其中任务的组织关系如下图：

`pull_rt_task` 的大概示意图如下：

- 当前CPU上的优先级任务不高，从另一个CPU的 `pushable_tasks` 链表中找优先级更高的任务来执行；

3.4.2 enqueue_task_rt/dequeue_task_rt

当RT任务进行出队入队时，通过 `enqueue_task_rt/dequeue_task_rt` 两个接口来完成，调用流程如下：

- `enqueue_task_rt` 和 `dequeue_task_rt` 都会调用 `dequeue_rt_stack` 接口，当请求的rt_se对应的是任务组时，会从顶部到请求的rt_se将调度实体出列；
- 任务添加到rt运行队列时，如果存在多个任务可以分配给多个CPU，设置overload，用于任务的迁移；

有点累了，收工了。

LoyenWang

喜欢作者

2 人喜欢