



Linux进程调度器的设计--Linux进程的管理与调度(十七)

时间：2016-07-21 作者：admin 分类：新手入门 (/emb-linux/entry-level/) 阅读:125次

日期	内核版本	架构	作者	GitHub	CSDN
2016-06-14	Linux-4.6	X86 & arm	gatieme	LinuxDeviceDrivers	Linux进程管理与调度

前景回顾

进程调度

内存中保存了对每个进程的唯一描述, 并通过若干结构与其他进程连接起来.

调度器面对的情形就是这样, 其任务是在程序之间共享CPU时间, 创造并行执行的错觉, 该任务分为两个不同的部分, 其中一个涉及**调度策略**, 另外一个涉及**上下文切换**.

内核必须提供一种方法, 在各个进程之间尽可能公平地共享CPU时间, 而同时又要考虑不同的任务优先级.

调度器的一个重要目标是有效地分配 CPU 时间片, 同时提供很好的用户体验. 调度器还需要面对一些互相冲突的目标, 例如既要为关键实时任务最小化响应时间, 又要最大限度地提高 CPU 的总体利用率.

调度器的一般原理是, 按所需分配的计算能力, 向系统中每个进程提供最大的公正性, 或者从另外一个角度上说, 他试图确保没有进程被亏待.

进程的分类

linux把进程区分为实时进程和非实时进程, 其中非实时进程进一步划分为交互式进程和批处理进程

类型	描述	示例
交互式进程 (interactive process)	此类进程经常与用户进行交互, 因此需要花费很多时间等待键盘和鼠标操作. 当接受了用户的输入后, 进程必须很快被唤醒, 否则用户会感觉系统反应迟钝	shell, 文本编辑程序和图形应用程序
批处理进程 (batch process)	此类进程不必与用户交互, 因此经常在后台运行. 因为这样的进程不必很快相应, 因此常受到调度程序的怠慢	程序语言的编译程序, 数据库搜索引擎以及科学计算
实时进程(real-time process)	这些进程由很强的调度需要, 这样的进程绝不会被低优先级的进程阻塞. 并且他们的响应时间要尽可能的短	视频音频应用程序, 机器人控制程序以及从物理传感器上收集数据的程序

在linux中, 调度算法可以明确的确认所有实时进程的身份, 但是没办法区分交互式程序和批处理程序, linux2.6的调度程序实现了基于进程过去行为的启发式算法, 以确定进程应该被当做交互式进程还是批处理进程. 当然与批处理进程相比, 调度程序有偏爱交互式进程的倾向

不同进程采用不同的调度策略

根据进程的不同分类Linux采用不同的调度策略.

对于实时进程，采用FIFO或者Round Robin的调度策略.

对于普通进程，则需要区分交互式和批处理式的不同。传统Linux调度器提高交互式应用的优先级，使得它们能更快地被调度。而CFS和RSDL等新的调度器的核心思想是“完全公平”。这个设计理念不仅大大简化了调度器的代码复杂度，还对各种调度需求的提供了更完美的支持.

注意Linux通过将进程和线程调度视为一个，同时包含二者。进程可以看做是单个线程，但是进程可以包含共享一定资源（代码和/或数据）的多个线程。因此进程调度也包含了线程调度的功能.

目前非实时进程的调度策略比较简单, 因为实时进程值只要求尽可能快的被响应, 基于优先级, 每个进程根据它重要程度的不同被赋予不同的优先级, 调度器在每次调度时, 总选择优先级最高的进程开始执行. 低优先级不可能抢占高优先级, 因此FIFO或者Round Robin的调度策略即可满足实时进程调度的需求.

但是普通进程的调度策略就比较麻烦了, 因为普通进程不能简单的只看优先级, 必须公平的占有CPU, 否则很容易出现进程饥饿, 这种情况下用户会感觉操作系统很卡, 响应总是很慢, 因此在linux调度器的发展历程中经过了多次重大变动, linux总是希望寻找一个最接近于完美的调度策略来公平快速的调度进程.

linux调度器的演变

一开始的调度器是复杂度为O(n)O(n)的始调度算法(实际上每次会遍历所有任务，所以复杂度为O(n)), 这个算法的缺点是当内核中有很多任务时，调度器本身就会耗费不少时间，所以，从linux2.5开始引入赫赫有名的O(1)O(1)调度器

然而，linux是集全球很多程序员的聪明才智而发展起来的超级内核，没有最好，只有更好，在O(1)O(1)调度器风光了没几天就被另一个更优秀的调度器取代了，它就是CFS调度器Completely Fair Scheduler. 这个也是在2.6内核中引入的，具体为2.6.23，即从此版本开始，内核使用CFS作为它的默认调度器，O(1)O(1)调度器被抛弃了, 其实CFS的发展也是经历了很多阶段，最早期的楼梯算法(SD), 后来逐步对SD算法进行改进出RSDL(Rotating Staircase Deadline Scheduler), 这个算法已经是“完全公平”的雏形了，直至CFS是最终被内核采纳的调度器, 它从RSDL/SD中吸取了完全公平的思想，不再跟踪进程的睡眠时间，也不再企图区分交互式进程。它将所有的进程都统一对待，这就是公平的含义。CFS的算法和实现都相当简单，众多的测试表明其性能也非常优越

字段	版本
O(n)的始调度算法	linux-0.11~2.4
O(1)调度器	linux-2.5
CFS调度器	linux-2.6~至今

Linux的调度器组成

2个调度器

可以用两种方法来激活调度

一种是直接的, 比如进程打算睡眠或出于其他原因放弃CPU

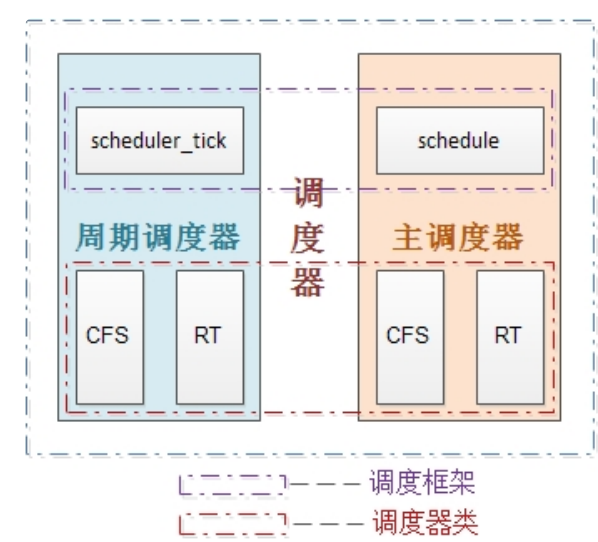
另一种是通过周期性的机制, 以固定的频率运行, 不时的检测是否有必要

因此当前linux的调度程序由两个调度器组成：**主调度器**，**周期性调度器**(两者又统称为**通用调度器(generic scheduler)**或**核心调度器(core scheduler)**)

并且每个调度器包括两个内容：**调度框架**(其实质就是两个函数框架)及**调度器类**

调度器类是实现了不同调度策略的实例，如 CFS、RT class等。

它们的关系如下图




当前的内核支持两种调度器类（ sched_setscheduler系统调用可修改进程的策略 ）：CFS（公平）、RT（实时）；5种调度策略：SCHED_NORAML（最常见的策略）、SCHED_BATCH（除了不能抢占外与常规任务一样，允许任务运行更长时间，更好地使用高速缓存，适合于成批处理的工作）、SCHED_IDLE（它甚至比nice 19还有弱，为了避免优先级反转使用）和SCHED_RR（循环调度，拥有时间片，结束后放在队列末）、SCHED_FIFO（没有时间片，可以运行任意长的时间）；其中前面三种策略使用的是cfs调度器类，后面两种使用rt调度器类。

2个调度器类

当前的内核支持2种调度器类（ sched_setscheduler系统调用可修改进程的策略 ）：**CFS（公平调度器）、RT（实时调度器）**

5种调度策略

字段	描述	所在调度器类
SCHED_NORMAL	（也叫SCHED_OTHER）用于普通进程，通过CFS调度器实现。SCHED_BATCH用于非交互的处理器消耗型进程。SCHED_IDLE是在系统负载很低时使用	CFS
SCHED_BATCH	SCHED_NORMAL普通进程策略的分化版本。采用分时策略，根据动态优先级(可用nice()API设置)，分配CPU运算资源。注意：这类进程比上述两类实时进程优先级低，换言之，在有实时进程存在时，实时进程优先调度。但针对吞吐量优化，除了不能抢占外与常规任务一样，允许任务运行更长时间，更好地使用高速缓存，适合于成批处理的工作	CFS

 SCHED_IDLE	优先级最低，在系统空闲时才跑这类进程(如利用闲散计算机资源跑地外文明搜索，蛋白质结构分析等任务，是此调度策略的适用者)	CFS
SCHED_FIFO	先入先出调度算法（实时调度策略），相同优先级的任务先到先服务，高优先级的任务可以抢占低优先级的任务	RT
SCHED_RR	轮流调度算法（实时调度策略），后者提供 Round-Robin 语义，采用时间片，相同优先级的任务当用完时间片会被放到队列尾部，以保证公平性，同样，高优先级的任务可以抢占低优先级的任务。不同要求的实时任务可以根据需要用 sched_setscheduler()API 设置策略	RT
SCHED_DEADLINE	新支持的实时进程调度策略，针对突发型计算，且对延迟和完成时间高度敏感的任务适用。基于Earliest Deadline First (EDF)最早截止期限优先调度算法	DL

最早截止时间优先（Earliest Deadline First，EDF）是实时系统中常用的一种调度算法，系统中有多个任务时，由调度算法决定哪个任务当前占用处理器，那么EDF算法就是按照任务的截止时间（deadline）来确定任务的执行顺序，最早截止的任务先执行。

最早截止时间优先EDF（Earliest DeadlineFirst）算法是非常著名的实时调度算法之一。在每一个新的就绪状态，调度器都是从那些已就绪但还没有完全处理完毕的任务中选择最早截止时间的任务，并将执行该任务所需的资源分配给它。在有新任务到来时，调度器必须立即计算EDF，排出新的定序，即正在运行的任务被剥夺，并且按照新任务的截止时间决定是否调度该新任务。如果新任务的最后期限早于被中断的当前任务，就立即处理新任务。按照EDF算法，被中断任务的处理将在稍后继续进行。

该算法的思想是从两个任务中选择截至时间最早的任务,把它暂作为当前处理任务,再判断该任务是否在当前周期内,若不在当前周期内,就让另一任务暂作当前处理任务,若该任务也不在当前周期内,就让CPU空跑到最靠近的下一个截至时间的开始,若有任务在该周期内,就判断该任务的剩余时间是否小于当前截至时间与当前时间的差,若小于,则让该任务运行到结束.否则,就让该任务运行到该周期的截止时间,就立即抢回处理器,再判断紧接着的最早截至时间,并把处理器给它,做法同上,如此反复执行.

其中前面三种策略使用的是cfs调度器类，后面两种使用rt调度器类。

另外，对于调度框架及调度器类，它们都有自己管理的运行队列，调度框架只识别rq（其实它也不能算是运行队列），而对于cfs调度器类它的运行队列则是cfs_rq（内部使用红黑树组织调度实体），实时rt的运行队列则为rt_rq（内部使用优先级bitmap+双向链表组织调度实体）

本质上, 通用调度器(核心调度器)是一个分配器,与其他两个组件交互.

调度器用于判断接下来运行哪个进程.

内核支持不同的调度策略(完全公平调度, 实时调度, 在无事可做的时候调度空闲进程,即0号进程也叫swapper进程,idle进程), 调度类使得能够以模块化的方法实现这些侧露额, 即一个类的代码不需要与其他类的代码交互

当调度器被调用时, 他会查询调度器类, 得知接下来运行哪个进程

在选中将要运行的进程之后, 必须执行底层的任务切换.

这需要与CPU的紧密交互. 每个进程刚好属于某一调度类, 各个调度类负责管理所属的进程. 通用调度器自身不涉及进程管理, 其工作都委托给调度器类.

进程调度的数据结构

调度器使用一系列数据结构来排序和管理系统中的进程. 调度器的工作方式的这些结构的涉及密切相关, 几个组件在许多方面

task_struct中调度相关的成员

```
[code]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;
    struct sched_dl_entity dl;

#ifdef CONFIG_CGROUP_SCHED
    /* 指向其所在进程组 */
    struct task_group *sched_task_group;
#endif
    .....
}
```

优先级

```
[code]int prio, static_prio, normal_prio;
unsigned int rt_priority;
```

动态优先级 静态优先级 实时优先级

其中task_struct采用了三个成员表示进程的优先级:prio和normal_prio表示动态优先级, static_prio表示进程的静态优先级.

为什么表示动态优先级需要两个值prio和normal_prio

调度器会考虑的优先级则保存在prio. 由于在某些情况下内核需要暂时提高进程的优先级, 因此需要用prio表示. 由于这些改变不是持久的, 因此静态优先级static_prio和普通优先级normal_prio不受影响.

此外还用了一个字段rt_priority保存了实时进程的优先级

字段	描述
static_prio	用于保存静态优先级, 是进程启动时分配的优先级, , 可以通过nice和sched_setscheduler系统调用来进行修改, 否则在进程运行期间会一直保持恒定
prio	保存进程的动态优先级
normal_prio	表示基于进程的静态优先级static_prio和调度策略计算出的优先级. 因此即使普通进程和实时进程具有相同的静态优先级, 其普通优先级也是不同的, 进程分叉(fork)时, 子进程会继承父进程的普通优先级
rt_priority	用于保存实时优先级

实时进程的优先级用实时优先级rt_priority来表示

linux2.6内核将任务优先级进行了一个划分, 实时优先级范围是0到MAX_RT_PRIO-1 (即99) , 而普通进程的静态优先级范围是从MAX_RT_PRIO到MAX_PRIO-1 (即100到139) 。

```
[code]/* */
#define MAX_USER_RT_PRIO    100
#define MAX_RT_PRIO        MAX_USER_RT_PRIO

/* */
#define MAX_PRIO            (MAX_RT_PRIO + 40)
#define DEFAULT_PRIO        (MAX_RT_PRIO + 20)
```

优先级范围	描述
0——99	实时进程
100——139	非实时进程

调度策略

```
[code]unsigned int policy;
```

policy保存了进程的调度策略，目前主要有以下五种：

```
≡[code]/*
 * Scheduling policies
 */
#define SCHED_NORMAL      0
#define SCHED_FIFO        1
#define SCHED_RR          2
#define SCHED_BATCH       3
/* SCHED_ISO: reserved but not implemented yet */
#define SCHED_IDLE        5
#define SCHED_DEADLINE    6
```

字段	描述	所在调度器类
SCHED_NORMAL	(也叫SCHED_OTHER)用于普通进程，通过CFS调度器实现。	
SCHED_BATCH	SCHED_NORMAL普通进程策略的分化版本。采用分时策略，根据动态优先级(可用nice()API设置)，分配CPU运算资源。注意：这类进程比两类实时进程优先级低，换言之，在有实时进程存在时，实时进程优先调度。但针对吞吐量优化	CFS
SCHED_IDLE	优先级最低，在系统空闲时才跑这类进程(如利用闲散计算机资源跑地外文明搜索，蛋白质结构分析等任务，是此调度策略的适用者)	CFS
SCHED_FIFO	先入先出调度算法(实时调度策略)，相同优先级的任务先到先服务，高优先级的任务可以抢占低优先级的任务	RT
SCHED_RR	轮流调度算法(实时调度策略)，后者提供Round-Robin语义，采用时间片，相同优先级的任务当用完时间片会被放到队列尾部，以保证公平性，同样，高优先级的任务可以抢占低优先级的任务。不同要求的实时任务可以根据需要用sched_setscheduler()API设置策略	RT
SCHED_DEADLINE	新支持的实时进程调度策略，针对突发型计算，且对延迟和完成时间高度敏感的任务适用。基于Earliest Deadline First (EDF) 调度算法	

CHED_BATCH用于非交互的处理器消耗型进程

CHED_IDLE是在系统负载很低时使用CFS

SCHED_BATCH用于非交互，CPU使用密集型的批处理进程。调度决策对此类进程给予“冷处理”：他们绝不会抢占CF调度器处理的另一个进程，因此不会干扰交互式进程。如果打算使用nice值降低进程的静态优先级，同时又不希望该进程影响系统的交互性，此时最适合使用该调度类。

而SCHED_LDLE进程的重要性则会进一步降低，因此其权重总是最小的

注意

尽管名称是SCHED_IDLE但是SCHED_IDLE不负责调度空闲进程。空闲进程由内核提供单独的机制来处理

SCHED_RR和SCHED_FIFO用于实现软实时进程。SCHED_RR实现了轮流调度算法，一种循环时间片的方法，而SCHED_FIFO实现

了先进先出的机制, 这些并不是由完全贡品调度器类CFS处理的, 而是由实时调度类处理.

三

调度策略相关字段

```
[code]
unsigned int policy;

const struct sched_class *sched_class;
struct sched_entity se;
struct sched_rt_entity rt;
struct sched_dl_entity dl;

cpumask_t cpus_allowed;
```

字段	描述
sched_class	调度类, 调度类, 调度处理函数类
se	普通进程的调用实体, 每个进程都有其中之一的实体
rt	实时进程的调用实体, 每个进程都有其中之一的实体
dl	deadline的调度实体
cpus_allowed	用于控制进程可以在哪里处理器上运行

调度器不限于调度进程, 还可以调度更大的实体, 比如实现组调度: 可用的CPU时间首先在一半的进程组(比如, 所有进程按照所有者分组)之间分配, 接下来分配的时间再在组内进行二次分配

cpus_allows是一个位域, 在多处理器系统上使用, 用来限制进程可以在哪些CPU上运行

调度类

sched_class结构体表示调度类, 类提供了通用调度器和各个调度器之间的关联, 调度器类和特定数据结构中汇集地几个函数指针表示, 全局调度器请求的各个操作都可以用一个指针表示, 这使得无需了解调度器类的内部工作原理即可创建通用调度器, 定义在kernel/sched/sched.h


```

≡[code]struct sched_class {
    /* 系统中多个调度类，按照其调度的优先级排成一个链表
    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);
    /* 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);

    /*
     * The switched_from() call is allowed to drop rq->lock, therefore we
     * cannot assume the switched_from/switched_to pair is serialized by
     * rq->lock. They are however serialized by p->pi_lock.
     */
    /* 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);
#endif
};
```

成员	描述
enqueue_task	向就绪队列中添加一个进程, 某个任务进入可运行状态时, 该函数将得到调用。它将调度实体（进程）放入红黑树中, 并对nr_running 变量加 1
dequeue_task	将一个进程从就绪队列中删除, 当某个任务退出可运行状态时调用该函数, 它将从红黑树中去掉对应的调度实体, 并从nr_running 变量中减 1
yield_task	在进程想要资源放弃对处理器的控制权的时, 可使用在sched_yield系统调用, 会调用内核API yield_task完成此工作. 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
check_preempt_curr	该函数将检查当前运行的任务是否被抢占。在实际抢占正在运行的任务之前, CFS 调度程序模块将执行公平性测试。这将驱动唤醒式（wakeup）抢占
pick_next_task	该函数选择接下来要运行的最合适的进程
put_prev_task	用另一个进程代替当前运行的进程
set_curr_task	当任务修改其调度类或修改其任务组时, 将调用这个函数
task_tick	在每次激活周期调度器时, 由周期性调度器调用, 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
task_new	内核调度程序为调度模块提供了管理新任务启动的机会, 用于建立fork系统调用和调度器之间的关联, 每次新进程建立后, 则用new_task通知调度器, CFS 调度模块使用它进行组调度, 而用于实时任务的调度模块则不会使用这个函数

对于各个调度器类, 都必须提供struct sched_class的一个实例, 目前内核中有实现以下五种:

```
[code]// extern const struct sched_class stop_sched_class;
extern const struct sched_class dl_sched_class;
extern const struct sched_class rt_sched_class;
extern const struct sched_class fair_sched_class;
extern const struct sched_class idle_sched_class;
```

调度器类	定义	描述
------	----	----

≡ stop_sched_class	kernel/sched/stop_task.c, line 112	优先级最高的线程，会中断所有其他线程，且不会被其他任务打断。作用： 1.发生在cpu_stop_cpu_callback 进行cpu之间任务migration； 2.HOTPLUG_CPU的情况下关闭任务。
dl_sched_class	kernel/sched/deadline.c, line 1774	
rt_sched_class	kernel/sched/rt.c, line 2326	RT，作用：实时线程
idle_sched_class	kernel/sched/idle_task.c, line 81	每个cup的第一个pid=0线程：swapper，是一个静态线程。调度类属于：idel_sched_class，所以在ps里面是看不到的。一般运行在开机过程和cpu异常的时候做dump
fair_sched_class	kernel/sched/fair.c, line 8521	CFS（公平调度器），作用：一般常规线程

目前系統中,Scheduling Class的优先级顺序为

```
[code]stop_sched_class -> dl_sched_class -> rt_sched_class -> fair_sched_class -> idle_sched_class
```

开发者可以根据己的设计需求,來把所属的Task配置到不同的Scheduling Class中.

用户层应用程序无法直接与调度类交互, 他们只知道上下文定义的常量SCHED_XXX(用task_struct->policy表示), 这些常量提供了调度类之间的映射。

SCHED_NORMAL, SCHED_BATCH, SCHED_IDLE被映射到fair_sched_class

SCHED_RR和SCHED_FIFO则与rt_schedule_class相关联

就绪队列

就绪队列是核心调度器用于管理活动进程的主要数据结构。

各个CPU都有自身的就绪队列，各个活动进程只出现在一个就绪队列中, 在多个CPU上同时运行一个进程是不可能的.

早期的内核中就绪队列是全局的, 即即有全局唯一的rq, 但是 在Linux-2.6内核时代，为了更好的支持多核，Linux调度器普遍采用了per-cpu的run queue，从而克服了多CPU系统中，全局唯一的run queue由于资源的竞争而成为了系统瓶颈的问题，因为 在同一时刻，一个CPU访问run queue时，其他的CPU即使空闲也必须等待，大大降低了整体的CPU利用率和系统性能。当使用 per-CPU的run queue之后，每个CPU不再使用大内核锁，从而大大提高了并行处理的调度能力。

参照CFS调度的总结 - （单rq vs 多rq）

就绪队列是全局调度器许多操作的起点, 但是进程并不是由就绪队列直接管理的, 调度管理是各个调度器的职责, 因此在各个就绪队列中嵌入了特定调度类的子就绪队列(cfs的顶级调度就队列 struct cfs_rq, 实时调度类的就绪队列struct rt_rq和deadline调度类的就绪队列struct dl_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)。

CPU就绪队列struct rq

就绪队列用struct rq来表示, 其定义在kernel/sched/sched.h, line 566

```
≡[code] /*每个处理器都会配置一个rq*/
```

```
struct rq {
    /* runqueue lock: */
    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.
     */
    /*用以记录目前处理器rq中执行task的数量*/
    unsigned long nr_running;
#ifdef CONFIG_NUMA_BALANCING
    unsigned int nr_numa_running;
    unsigned int nr_preferred_running;
#endif

#define CPU_LOAD_IDX_MAX 5
    /*用以表示处理器的负载，在每个处理器的rq中都会有对应到该处理器的cpu_load参数配置，
    在每次处理器触发scheduler tick时，都会调用函数update_cpu_load_active,进行cpu_load的更新
    在系统初始化的时候会调用函数sched_init把rq的cpu_load array初始化为0.
    了解他的更新方式最好的方式是通过函数update_cpu_load,公式如下
    cpu_load[0]会直接等待rq中load.weight的值。
    cpu_load[1]=(cpu_load[1]*(2-1)+cpu_load[0])/2
    cpu_load[2]=(cpu_load[2]*(4-1)+cpu_load[0])/4
    cpu_load[3]=(cpu_load[3]*(8-1)+cpu_load[0])/8
    cpu_load[4]=(cpu_load[4]*(16-1)+cpu_load[0])/16
    调用函数this_cpu_load时，所返回的cpu load值是cpu_load[0]
    而在进行cpu blance或migration时，就会呼叫函数
    source_load target_load取得对该处理器cpu_load index值，
    来进行计算*/
    unsigned long cpu_load[CPU_LOAD_IDX_MAX];
    unsigned long last_load_update_tick;

#ifdef CONFIG_NO_HZ_COMMON
    u64 nohz_stamp;
    unsigned long nohz_flags;
#endif
#ifdef CONFIG_NO_HZ_FULL
    unsigned long last_sched_tick;
#endif

    /* capture load from *all* tasks on this cpu: */
    /*load->weight值，会是目前所执行的schedule entity的load->weight的总和
    也就是说rq的load->weight越高，也表示所负责的排程单元load->weight总和越高
    表示处理器所负荷的执行单元也越重*/
    struct load_weight load;
    /*在每次scheduler tick中呼叫update_cpu_load时，这个值就增加一，
    可以用来反馈目前cpu load更新的次数*/
    unsigned long nr_load_updates;
    /*用来累加处理器进行context switch的次数，会在调用schedule时进行累加，
    并可以通过函数nr_context_switches统计目前所有处理器总共的context switch次数
    或是可以透过查看档案/proc/stat中的ctxtt位得知目前整个系统触发context switch的次数*/
    u64 nr_switches;

    /*为cfs fair scheduling class 的rq就绪队列 */
    struct cfs_rq cfs;
    /*为real-time scheduling class 的rq就绪队列 */
    struct rt_rq rt;
    /* 为deadline scheduling class 的rq就绪队列 */

    /* 用以支援可以group cfs tasks的机制*/
#ifdef CONFIG_FAIR_GROUP_SCHED
```



```

/* list of leaf cfs_rq on this cpu: */
/*
在有设置fair group scheduling 的环境下,
会基于原本cfs_rq中包含有若干task的group所成的排程集合,
也就是说当有一个group a就会有自己的cfs_rq用来排程自己所属的tasks,
而属于这group a的tasks所使用到的处理器时间就会以这group a总共所分的的时间为上限。
基于cgroup的fair group scheduling 架构, 可以创造出有阶层性的task组织,
根据不同task的功能群组化在配置给该群主对应的处理器资源,
让属于该群主下的task可以透过rq机制使用该群主下的资源。
这个变数主要是管理CFS RQ list,
操作上可以透过函数list_add_leaf_cfs_rq把一个group cfs_rq加入到list中,
或透过函数list_del_leaf_cfs_rq把一个group cfs_rq移除,
并可以透过for_each_leaf_cfs_rq把一个rq上得所有leaf cfs_rq走一遍
*/
struct list_head leaf_cfs_rq_list;
#endif
/*
 * 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:
 */
/*一般来说, linux kernel 的task状态可以为
TASK_RUNNING, TASK_INTERRUPTIBLE(sleep), TASK_UNINTERRUPTIBLE(Deactivate Task),
此时Task会从rq中移除)或TASK_STOPPED.
透过这个变量会统计目前rq中有多少task属于TASK_UNINTERRUPTIBLE的状态。
当调用函数active_task时, 会把nr_uninterruptible值减一,
并透过该函数enqueue_task把对应的task依据所在的scheduling class放在对应的rq中
并把目前rq中nr_running值加一 */
unsigned long nr_uninterruptible;

/*
curr:指向日前处理器正在执行的task;
idle:指向属于idle-task scheduling class 的idle task;
stop:指向日前最高等级属于stop-task scheduling class
的task; */
struct task_struct *curr, *idle;
/*
基于处理器的jiffies值, 用以记录下次进行处理器balancing 的时间点*/
unsigned long next_balance;
/*
用以存储context-switch发生时,
前一个task的memory management结构并可用在函数finish_task_switch
透过函数mmdrop释放前一个task的结构体资源 */
struct mm_struct *prev_mm;

unsigned int clock_skip_update;

/* 用以记录目前rq的clock值,
基本上该值会等于通过sched_clock_cpu(cpu_of(rq))的返回值,
并会在每次调用scheduler_tick时通过函数update_rq_clock更新目前rq clock值。
函数sched_clock_cpu会通过sched_clock_local或ched_clock_remote取得
对应的sched_clock_data,而处理的sched_clock_data值,
会通过函数sched_clock_tick在每次调用scheduler_tick时进行更新;
*/
u64 clock;
u64 clock_task;

/*用以记录目前rq中有多少task处于等待i/o的sleep状态
在实际的使用上, 例如当driver接受来自task的调用,
但处于等待i/o回复的阶段时, 为了充分利用处理器的执行资源,
这时就可以在driver中调用函数io_schedule,

```



此时就会把目前rq中的nr_iowait加一, 并设定目前task的io_wait为1
 然后触发scheduling 让其他task有机会可以得到处理器执行时间*/
 atomic_t nr_iowait;

```
#ifdef CONFIG_SMP
/*root domain是基于多核心架构下的机制,
  会由rq结构记住目前采用的root domain,
  其中包括了目前的cpu mask(包括span,online rt overload), reference count 跟cpupri
  当root domain有被rq参考到时, refcount 就加一, 反之就减一。
  而cpumask span表示rq可挂上的cpu mask,noline为rq目前已经排程的
  cpu mask cpu上执行real-time task.可以参考函数pull_rt_task, 当一个rq中属于
  real-time的task已经执行完毕, 就会透过函数pull_rt_task从该
  rq中属于rto_mask cpu mask 可以执行的处理器上, 找出是否有一个处理器
  有大于一个以上的real-time task, 若有就会转到目前这个执行完成
  real-time task 的处理器上
  而cpupri不同于Task本身有区分140个(0-139)
  Task Priority (0-99为RT Priority 而 100-139为Nice值 -20-19).
  CPU Priority本身有102个Priority (包括,-1为Invalid,
  0为Idle,1为Normal,2-101对应到Real-Time Priority 0-99).
  参考函数convert_prio, Task Priority如果是 140就会对应到
  CPU Idle,如果是>=100就会对应到CPU Normal,
  若是Task Priority介于0-99之间,就会对应到CPU Real-Time Priority 101-2之间.)
  在实际的操作上,例如可以通过函数cpupri_find 传入一个要插入的Real-Time Task,
  此时就会依据cpupri中pri_to_cpu选择一个目前执行Real-Time Task
  且该Task的优先级比目前要插入的Task更低的处理器,
  并通过CPU Mask(lowest_mask)返回目前可以选择的处理器Mask.
  可以参考kernel/sched_cpupri.c.
  在初始化的过程中,通过函数sched_init调用函数init_defrootdomain,
  对Root Domain和CPU Priority机制进行初始化.
  */
struct root_domain *rd;

/*Schedule Domain是基于多核心架构下的机制.
  每个处理器都会有一个基础的Scheduling Domain,
  Scheduling Domain可以通过parent找到上一层的Domain,
  或是通过child找到下一层的 Domain (NULL表示结尾.).
  也可以通过span字段,表示这个Domain所能覆盖的处理器范围.
  通常Base Domain会涵盖系统中所有处理器的个数,
  而Child Domain所能涵盖的处理器个数不超过它的Parent Domain.
  而当进行Scheduling Domain 中的Task Balance,就会以该Domain所涵盖的处理器为最大范围.
  同时,每个Schedule Domain都会包括一个或一个以上的
  CPU Groups (结构为struct sched_group),并通过next字段把
  CPU Groups链接在一起(成为一个单向的Circular linked list),
  每个CPU Group都会有变量cpumask来定义CPU Group
  可以参考Linux Kernel文件 Documentation/scheduler/sched-domains.txt.
  */
struct sched_domain *sd;

struct callback_head *balance_callback;

unsigned char idle_balance;
/* For active balancing */
int active_balance;
int push_cpu;
struct cpu_stop_work active_balance_work;
/* cpu of this runqueue: */
int cpu;
int online;

/*当RunQueue中此值为1,表示这个RunQueue正在进行
  Fair Scheduling的Load Balance,此时会调用stop_one_cpu_nowait
  暂停该RunQueue所出处理器调度,
```



```

并通过函数active_load_balance_cpu_stop,
把Tasks从最忙碌的处理器移到Idle的处理器上执行。  */
int active_balance;

/*用以存储目前进入Idle且负责进行Load Balance的处理器ID.
调用的流程为,在调用函数schedule时,
若该处理器RunQueue的nr_running为0 (也就是目前没有
正在执行的Task),就会调用idle_balance,并触发Load Balance  */
int push_cpu;
/* cpu of this runqueue: */
/*用以存储前运作这个RunQueue的处理器ID*/
int cpu;

/*为1表示目前此RunQueue有在对应的处理器上并执行  */
int online;

/*如果RunQueue中目前有Task正在执行,
这个值会等等于该RunQueue的Load Weight除以目前RunQueue中Task数目的均值.
(rq->avg_load_per_task = rq->load.weight / nr_running;).*/
unsigned long avg_load_per_task;

/*这个值会由Real-Time Scheduling Class调用函数update_curr_rt,
用以统计目前Real-Time Task执行时间的均值,
在这个函数中会以目前RunQueue的clock_task减去目前Task执行的起始时间,
取得执行时间的Delta值. (delta_exec = rq->clock_task - curr->se.exec_start; ).
在通过函数sched_rt_avg_update把这个Delta值跟原本RunQueue中的rt_avg值取平均值.
以运行的周期来看,这个值可反应目前系统中Real-Time Task平均被分配到的执行时间值  .*/
u64 rt_avg;

/* 这个值主要在函数sched_avg_update更新  */
u64 age_stamp;

/*这值会在处理Scheduling时,若判断目前处理器runQueue没有正在运行的Task,
就会通过函数idle_balance更新这个值为目前RunQueue的clock值.
可用以表示这个处理器何时进入到Idle的状态  */
u64 idle_stamp;

/*会在有Task运行且idle_stamp不为0 (表示前一个转台是在Idle)时
以目前RunQueue的clock减去idle_stmp所计算出的Delta值为依据,
更新这个值, 可反应目前处理器进入Idle状态的时间长短  */
u64 avg_idle;

/* This is used to determine avg_idle's max value */
u64 max_idle_balance_cost;
#endif

#ifdef CONFIG_IRQ_TIME_ACCOUNTING
    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 */
/*用以记录下一次计算CPU Load的时间,
初始值为目前的jiffies加上五秒与1次的Scheduling Tick的间隔
(=jiffies + LOAD_FREQ,且LOAD_FREQ=(5*HZ+1))*
/
unsigned long calc_load_update;

```




```

/*等于RunQueue中nr_running与nr_uninterruptible的总和.
(可参考函式calc_load_fold_active).*/
long calc_load_active;

#ifdef CONFIG_SCHED_HRTICK
#ifdef CONFIG_SMP
    int hrtick_csd_pending;
    /*在函数it_rq_hrtick初始化RunQueue High-Resolution
    Tick时, 此值设为0.
    在函数hrtick_start中,会判断目前触发的RunQueue跟目前处理器所使用的RunQueue是否一致,
    若是,就直接呼叫函数hrtimer_restart,反之就会依据RunQueue中hrtick_csd_pending的值,
    如果hrtick_csd_pending为0,就会通过函数__smp_call_function_single让RunQueue所在的一个
    处理器执行rq->hrtick_csd.func和函数 __hrtick_start.
    并等待该处理器执行完毕后,才重新把hrtick_csd_pending设定为1.
    也就是说, RunQueue的hrtick_csd_pending是用来作为SMP架构下,
    由处理器A触发处理器B执行*/
    struct call_single_data hrtick_csd;
#endif
    /*为gh-Resolution Tick的结构,会通过htimer_init初始化.*/
    struct hrtimer hrtick_timer;
#endif

#ifdef CONFIG_SCHEDSTATS
    /* latency stats */
    /*為Scheduling Info.的統計結構,可以參考
    include/linux/sched.h中的宣告. 例如在每次觸發
    Schedule時,呼叫函式schedule_debug對上一個Task
    的lock_depth進行確認(Fork一個新的Process 時,
    會把此值預設為-1就是No-Lock,當呼叫
    Kernel Lock時, 就會把Current Task的lock_depth加一.),
    若lock_depth>=0,就會累加Scheduling Info.的bkl_count值,
    用以代表Task Blocking的次數.*/
    struct sched_info rq_sched_info;
    /*可用以表示RunQueue中的Task所得到CPU執行
    時間的累加值.
    在發生Task Switch時,會透過sched_info_switch呼叫
    sched_info_arrive並以目前RunQueue Clock值更新
    Task 的sched_info.last_arrival時間,而在Task所分配時間
    結束後,會在函式sched_info_depart中以現在的
    RunQueue Clock值減去Task的sched_info.last_arrival
    時間值,得到的 Delta作為變數rq_cpu_time的累
    加值.*/
    unsigned long long rq_cpu_time;
    /* could above be rq->cfs_rq.exec_clock + rq->rt_rq.rt_runtime ? */

    /* sys_sched_yield() stats */
    /*用以統計呼叫System Call sys_sched_yield的次數.*/
    unsigned int yld_count;

    /* schedule() stats */
    /*可用以統計觸發Scheduling的次數. 在每次觸發
    Scheduling時,會透過函式schedule呼叫schedule_debug,
    呼叫schedstat_inc對這變數進行累加.*/
    unsigned int sched_count;
    /*可用以統計進入到Idle Task的次數. 會在函式
    pick_next_task_idle中,呼叫schedstat_inc對這變數進行
    累加.*/
    unsigned int sched_goidle;

    /* try_to_wake_up() stats */
    /*用以統計Wake Up Task的次數.*/

```

≡

```
    unsigned int ttwu_count;
    /*用以統計Wake Up 同一個處理器Task的次數.*/
    unsigned int ttwu_local;

    /* BKL stats */
    unsigned int bkl_count;
#endif

#ifdef CONFIG_SMP
    struct llist_head wake_list;
#endif

#ifdef CONFIG_CPU_IDLE
    /* Must be inspected within a rcu lock section */
    struct cpuidle_state *idle_state;
#endif
};
```

字段	描述
nr_running	队列上可运行进程的数目, 不考虑优先级和调度类
load	提供了就绪队列当前负荷的度量, 队列的符合本质上与队列上当前活动进程的数目成正比, 其中的各个进程又有优先级作为权重. 每个就绪队列的虚拟时钟的速度等于该信息
cpu_load	用于跟踪此前的负荷状态
cfs,rt 和dl	嵌入的子就绪队列, 分别用于完全公平调度器, 实时调度器和deadline调度器
curr	当前运行的进程的task_struct实例
idle	指向空闲进程的task_struct实例
clock	就绪队列自身的时钟

系统中所有的就绪队列都在runqueues数组中, 该数组的每个元素分别对应于系统中的一个CPU, 如果是单处理器系统只有一个就绪队列, 则数组就只有一个元素

内核中也提供了一些宏, 用来获取cpu上的就绪队列的信息

```
[code]// DECLARE_PER_CPU_SHARED_ALIGNED(struct rq, runqueues);

#define cpu_rq(cpu)          (&per_cpu(runqueues, (cpu)))
#define this_rq()            this_cpu_ptr(&runqueues)
#define task_rq(p)           cpu_rq(task_cpu(p))
#define cpu_curr(cpu)        (cpu_rq(cpu)->curr)
#define raw_rq()             raw_cpu_ptr(&runqueues)
```

CFS公平调度器的就绪队列cfs_rq

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。

其定义在`kernel/sched/sched.h`#L359

```

≡[code]/* CFS-related fields in a runqueue */
/* 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 */
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;

    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;
#ifdef 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
     * 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
     */
    struct sched_avg avg;
    u64 runnable_load_sum;
    unsigned long runnable_load_avg;
#ifdef CONFIG_FAIR_GROUP_SCHED
    unsigned long tg_load_avg_contrib;
#endif
    atomic_long_t removed_load_avg, removed_util_avg;
#endif
#ifdef 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 lrgs 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;
    /* 拥有该CFS运行队列的进程组 */
    struct task_group *tg; /* group that "owns" this runqueue */

#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;
    struct list_head throttled_list;
#endif /* CONFIG_CFS_BANDWIDTH */
#endif /* CONFIG_FAIR_GROUP_SCHED */
};

```

实时进程就绪队列rt_rq

其定义在kernel/sched/sched.h#L449

```

≡[code]/* Real-Time classes' related field in a runqueue: */
struct rt_rq {
    struct rt_prio_array active;
    unsigned int rt_nr_running;
    unsigned int rr_nr_running;
#ifdef CONFIG_SMP || defined CONFIG_RT_GROUP_SCHED
    struct {
        int curr; /* highest queued rt task prio */
#ifdef CONFIG_SMP
        int next; /* next highest */
#endif
    } highest_prio;
#ifdef CONFIG_SMP
    unsigned long rt_nr_migratory;
    unsigned long rt_nr_total;
    int overloaded;
    struct plist_head pushable_tasks;
#endif
#ifdef HAVE_RT_PUSH_IPI
    int push_flags;
    int push_cpu;
    struct irq_work push_work;
    raw_spinlock_t push_lock;
#endif
#endif /* CONFIG_SMP */
    int rt_queued;

    int rt_throttled;
    u64 rt_time;
    u64 rt_runtime;
    /* Nests inside the rq lock: */
    raw_spinlock_t rt_runtime_lock;

#ifdef CONFIG_RT_GROUP_SCHED
    unsigned long rt_nr_boosted;

    struct rq *rq;
    struct task_group *tg;
#endif
};

```

EDF就绪队列dl_rq

其定义在kernel/sched/sched.h#L490

```

≡[code]/* Deadline class' related fields in a runqueue */
struct dl_rq {
    /* runqueue is an rbtree, ordered by deadline */
    struct rb_root rb_root;
    struct rb_node *rb_leftmost;

    unsigned long dl_nr_running;

#ifdef CONFIG_SMP
    /*
     * Deadline values of the currently executing and the
     * earliest ready task on this rq. Caching these facilitates
     * the decision whether or not a ready but not running task
     * should migrate somewhere else.
     */
    struct {
        u64 curr;
        u64 next;
    } earliest_dl;

    unsigned long dl_nr_migratory;
    int overloaded;

    /*
     * Tasks on this rq that can be pushed away. They are kept in
     * an rb-tree, ordered by tasks' deadlines, with caching
     * of the leftmost (earliest deadline) element.
     */
    struct rb_root pushable_dl_tasks_root;
    struct rb_node *pushable_dl_tasks_leftmost;
#else
    struct dl_bw dl_bw;
#endif
};

```

调度实体

我们前面提到, 调度器不限于调度进程, 还可以调度更大的实体, 比如实现组调度: 可用的CPU时间首先在一半的进程组(比如, 所有进程按照所有者分组)之间分配, 接下来分配的时间再在组内进行二次分配.

这种一般性要求调度器不直接操作进程, 而是处理可调度实体, 因此需要一个通用的数据结构描述这个调度实体, 即`seched_entity`结构, 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`sched_entity`结构, 其定义在`include/linux/sched.h`, 如下:

`sched_entity`调度实体

```

≡[code]/* 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 average tracking.
     *
     * Put into separate cache line so it does not
     * collide with read-mostly values above.
     */
    struct sched_avg       avg ____cacheline_aligned_in_smp;
#endif
};

```

在struct sched_entity结构中, 值得我们注意的成员是

字段	描述
load	指定了权重, 决定了各个实体占队列总负荷的比重, 计算负荷权重是调度器的一项重任, 因为CFS所需的虚拟时钟的速度最终依赖于负荷, 权重通过优先级转换而成, 是vruntime计算的关键
run_node	调度实体在红黑树对应的结点信息, 使得调度实体可以在红黑树上排序
sum_exec_runtime	记录程序运行所消耗的CPU时间, 以用于完全公平调度器CFS
on_rq	调度实体是否在就绪队列上接受检查, 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)

* 在进程运行时, 我们需要记录消耗的CPU时间, 以用于完全公平调度器. sum_exec_runtime就用于该目的.

跟踪运行时间是由update_curr不断累积完成的. 内核中许多地方都会调用该函数, 例如, 新进程加入就绪队列时, 或者周期性调度器中. 每次调用时, 会计算当前时间和exec_start之间的差值, exec_start则更新到当前时间. 差值则被加到sum_exec_runtime.

在进程执行期间虚拟时钟上流逝的时间数量由vruntime统计

在进程被撤销时, 其当前sum_exec_runtime值保存到prev_sum_exec_runtime, 此后, 进程抢占的时候需要用到该数据, 但是注意, 在prev_sum_exec_runtime中保存了sum_exec_runtime的值, 而sum_exec_runtime并不会被重置, 而是持续单调增长

每个进程的task_struct中都嵌入了sched_entity对象, 所以进程是可调度的实体, 但是请注意, 其逆命一般是不正确的, 即可调度的实体不一定是进程.

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还有个进程组C, 原理都一样, 就是一个层次结构。

实时进程调度实体sched_rt_entity

其定义在include/linux/sched.h, 如下：

```
≡[code]struct sched_rt_entity {
    struct list_head run_list;
    unsigned long timeout;
    unsigned long watchdog_stamp;
    unsigned int time_slice;
    unsigned short on_rq;
    unsigned short on_list;

    struct sched_rt_entity *back;
#ifdef CONFIG_RT_GROUP_SCHED
    struct sched_rt_entity *parent;
    /* rq on which this entity is (to be) queued: */
    struct rt_rq          *rt_rq;
    /* rq "owned" by this entity/group: */
    struct rt_rq          *my_q;
#endif
};
```

EDF调度实体sched_dl_entity

其定义在include/linux/sched.h, 如下：

```

≡[code]struct sched_dl_entity {
    struct rb_node  rb_node;

    /*
     * Original scheduling parameters. Copied here from sched_attr
     * during sched_setattr(), they will remain the same until
     * the next sched_setattr().
     */
    u64 dl_runtime;          /* maximum runtime for each instance */
    u64 dl_deadline;         /* relative deadline of each instance */
    u64 dl_period;          /* separation of two instances (period) */
    u64 dl_bw;              /* dl_runtime / dl_deadline */

    /*
     * Actual scheduling parameters. Initialized with the values above,
     * they are continuously updated during task execution. Note that
     * the remaining runtime could be < 0 in case we are in overrun.
     */
    s64 runtime;            /* remaining runtime for this instance */
    u64 deadline;           /* absolute deadline for this instance */
    unsigned int flags;      /* specifying the scheduler behaviour */

    /*
     * Some bool flags:
     *
     * @dl_throttled tells if we exhausted the runtime. If so, the
     * task has to wait for a replenishment to be performed at the
     * next firing of dl_timer.
     *
     * @dl_boosted tells if we are boosted due to DI. If so we are
     * outside bandwidth enforcement mechanism (but only until we
     * exit the critical section);
     *
     * @dl_yielded tells if task gave up the cpu before consuming
     * all its available runtime during the last job.
     */
    int dl_throttled, dl_boosted, dl_yielded;

    /*
     * Bandwidth enforcement timer. Each -deadline task has its
     * own bandwidth to be enforced, thus we need one timer per task.
     */
    struct hrtimer dl_timer;
};

```

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%。

如果task_group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_group中的下一个被调度进程；相反，如果task_group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

其定义在kernel/sched/sched.h?v=4.6#L240, 如下：

```
[code]/* task group related information */
struct task_group {
    struct cgroup_subsys_state css;

#ifdef CONFIG_FAIR_GROUP_SCHED
    /* schedulable entities of this group on each cpu */
    struct sched_entity **se;
    /* runqueue "owned" by this group on each cpu */
    struct cfs_rq **cfs_rq;
    unsigned long shares;

#ifdef CONFIG_SMP
    /*
     * load_avg can be heavily contended at clock tick time, so put
     * it in its own cacheline separated from the fields above which
     * will also be accessed at each tick.
     */
    atomic_long_t load_avg ____cacheline_aligned;
#endif
#endif

#ifdef CONFIG_RT_GROUP_SCHED
    struct sched_rt_entity **rt_se;
    struct rt_rq **rt_rq;

    struct rt_bandwidth rt_bandwidth;
#endif

    struct rcu_head rcu;
    struct list_head list;

    struct task_group *parent;
    struct list_head siblings;
    struct list_head children;

#ifdef CONFIG_SCHED_AUTOGROUP
    struct autogroup *autogroup;
#endif

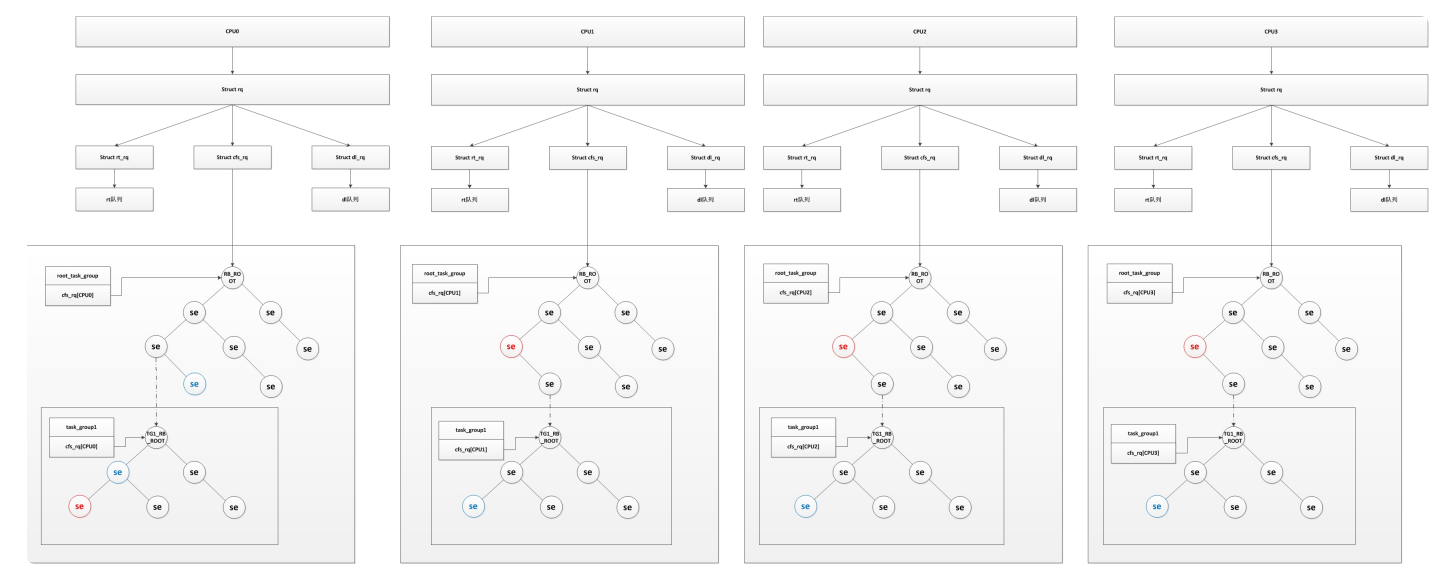
    struct cfs_bandwidth cfs_bandwidth;
};
```

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)。

总结

进程调度器的框架如下图所示



从图中可以看出来，每个CPU对应包含一个运行队列结构(struct rq)，而每个运行队列又包含有其自己的实时进程运行队列(struct rt_rq)、普通进程运行队列(struct cfs_rq)、和deadline实时调度的运行队列(struct dl_rq)，也就是说每个CPU都有他们自己的实时进程运行队列及普通进程运行队列

为了方便，我们在图中只描述普通进程的组织结构(最复杂的也是普通进程的组织结构)，而红色se则为当前CPU上正在执行的程序，蓝色为下个将要执行的程序，其实图中并不规范，实际上当进程运行时，会从红黑树中剥离出来，然后设定下一个调度进程，当进程运行时间结束时，再重新放入红黑树中。而为什么CPU0上有两个蓝色将被调度进程，将在组调度中解释。而为什么红黑树中又有一个子红黑树，我们将在调度实体中解释。

参照 linux调度器源码分析 - 概述(一)

通过的调度策略对象-调度类

linux下每个进程都由自身所属的调度类进行管理， sched_class结构体表示调度类, 调度类提供了通用调度器和各个调度器之间的关联, 调度器类和特定数据结构中汇集地几个函数指针表示, 全局调度器请求的各个操作都可以用一个指针表示, 这使得无需了解调度器类的内部工作原理即可创建通用调度器, 定义在kernel/sched/sched.h

开发者可以根据己的设计需求,来把所属的Task配置到不同的Scheduling Class中.

用户层应用程序无法直接与调度类交互, 他们只知道上下文定义的常量SCHED_XXX(用task_struct->policy表示), 这些常量提供了调度类之间的映射。

目前系统中,Scheduling Class的优先级顺序为

```
≡[code]stop_sched_class -> dl_sched_class -> rt_sched_class -> fair_sched_class -> idle_sched_class
```

被调度的实体-进程或者进程组

linux下被调度的不只是进程, 还可以是进程组. 因此需要一种更加通用的形式组织被调度数据结构, 即调度实体, 同样不同的进程用不同的调度实体表示

普通进程	实时进程
sched_entity	rt_entity, sched_dl_entity

用就绪队列保存和组织调度进程

所有的就绪进程(TASK_RUNNING)都被组织在就绪队列, 也叫运行队列中, 每个CPU对应包含一个运行队列结构(struct rq), 而每个运行队列又嵌入了有其自己的实时进程运行队列(struct rt_rq)、普通进程运行队列(struct cfs_rq)、和EDF实时调度的运行队列(struct dl_rq), 也就是说每个CPU都有他们自己的实时进程运行队列及普通进程运行队列

普通进程	实时进程
rq	rt_rq, dl_rq

本文永久更新链接 : <http://embeddedlinux.org.cn/emb-linux/entry-level/201607/21-5546.html> (/emb-linux/entry-level/201607/21-5546.html)

标签 :

上一篇 : [shell进行字符串截取的方法 \(/emb-linux/entry-level/201607/21-5545.html\)](#)

下一篇 : [linux---谈谈vfork和fork的区别及exit与return \(/emb-linux/entry-level/201607/21-5547.html\)](#)

相关推荐

- [linux环境变量设置方法总结 PATH、LD_LIBRARY_PATH \(/emb-linux/entry-level/201704/01-6456.html\)](#)
- [LaTeXila简介：Linux上的一个多语言LaTeX编辑器 \(/emb-linux/entry-level/201704/01-6455.html\)](#)
- [在Kali Linux中更改GRUB2背景的5种方式 \(/emb-linux/entry-level/201704/01-6454.html\)](#)
- [Linux基础之-正则表达式 \(grep , sed , awk \) \(/emb-linux/entry-level/201703/30-6355.html\)](#)
- [指南：在 Github 和 Git 上如何 Fork \(/emb-linux/entry-level/201703/30-6354.html\)](#)
- [在 Linux 桌面上使用 Gifine 录制 GIF 动画 \(/emb-linux/entry-level/201703/29-6346.html\)](#)
- [大神教你如何加入一个技术社区 \(/emb-linux/entry-level/201703/28-6344.html\)](#)

评论

