第 5 章　进程调度与负载均衡

调度工作涉及选择哪个（哪些）任务在哪个（哪些）处理器上运行，解决各个进程公平地享用 CPU 资源的问题。具体需要确定当前进程可以占用 CPU 核多久、哪个进程将是下一个要运行的进程。负载均衡主要解决的是各个 CPU 忙闲不一的问题，提高系统的整体吞吐率。

调度和负载均衡大体上是与硬件架构无关的，但是调度相关的进程切换则是体系结构紧密相关的内容（已在 4.3 节讨论），另外在考虑进程迁移的开销时需要考虑架构细节。

调度和负载均衡在图 1-5 的模型中属于进程管理的自主执行代码，用户仅能修改某些参数而不能直接干预调度和负载均衡操作。

本章内容相对独立，不像内存管理和文件系统那样互相关联，因此是比较容易理解的，大多数 Linux 内核分析的图书对这方面的论述也比较全面。

5.1　调度与均衡基本框架

就绪进程在 Linux 内核中的组织分成若干层次，首先是按不同处理器组织成不同的运行队列 rq（run queue），在每个处理器的 rq 内部又按紧急程度组织成 4 种“调度类”，各种调度类管理自己的就绪进程。各种调度类内部的进程间使用优先级进一步区分各自的重要程度。

进程调度解决的就是如何从这些就绪的进程中选择一个到 CPU 上去运行，而负载均衡就是在各个处理器上负载严重不均的时候将繁忙处理器 rq 上的进程迁移到空闲 CPU 的 rq 之上，因此 rq 是调度和负载均衡的基础数据结构，后面会展开讨论。上述工作可以简单地用图 5-1 表示。

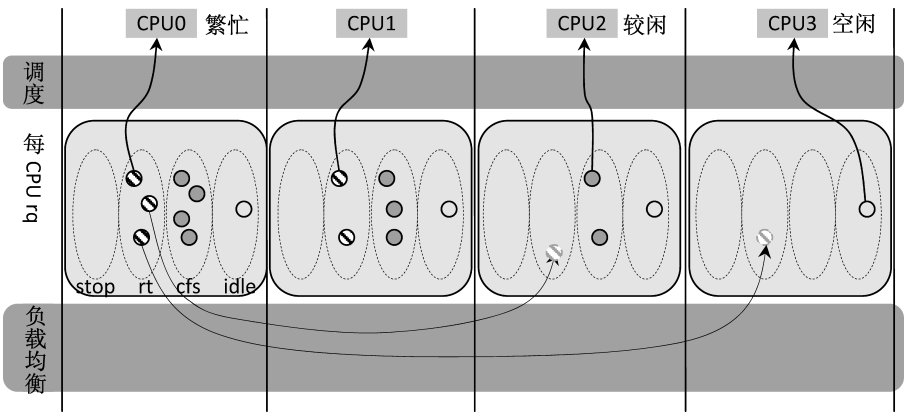

图 5-1　就绪进程组织与调度、均衡示意图

图 5-1 展示了 4 个 CPU 的系统上的例子，每个 CPU 有各自的 rq 就绪队列，每个 rq 管理着 4 种类型的任务——分别对应 STOP、RT、CFS 和 IDLE 调度类，优先级依次递减。实时任务属于 RT 调度类，普通进程属于 CFS 完全公平调度类。调度程序从各个 rq 数据结构中挑选一个进程作为各个 CPU 运行的任务，在合适的时机将现有任务撤下 CPU 而换上另一个更紧迫的任务。由于图 5-1 例子中没有 STOP 类进程，因此 CPU0/1 上执行的是实时 RT 类任务，而 CPU2 上执行的是普通 CFS 类任务，CPU3 上没有有效进程因此执行 IDLE 类任务。随着时间的推进，各个处理器会交替地执行本地的各个就绪进程。对于 CPU3 处于空闲 IDLE 状态，负载均衡器会将其他 CPU 上的就绪进程迁移过来，避免处理器的忙闲不一的状态。此例子中 CPU0 上的高优先级实时任务可通过负载均衡迁移到 CPU2/3 上。

5.2 进程状态与转换

在第 4 章讨论进程切换的时候，只涉及就绪进程。但实际上进程不仅区分为在 CPU 上运行和未获得 CPU 的两个基本状态，还有其他多种进程运行状态。

5.2.1 进程调度状态

Linux 是一个多用户、多任务的系统，可以同时运行多个用户的多个程序，就必然会产生很多的进程，而每个进程除了正在运行（拥有 CPU）外还会有其他不同的状态。这些状态的具体编码如代码 5-1 所示。

代码 5-1 进程状态 (linux-3.13/include/linux/sched.h)

```
135 #define TASK_RUNNING            0
136 #define TASK_INTERRUPTIBLE      1
137 #define TASK_UNINTERRUPTIBLE    2
138 #define __TASK_STOPPED          4
139 #define __TASK_TRACED           8
140 /* in tsk->exit_state */    以下两个状态出现在 task_struct-> exit_state
141 #define EXIT_ZOMBIE            16
142 #define EXIT_DEAD              32
143 /* in tsk->state again */
144 #define TASK_DEAD              64
145 #define TASK_WAKEKILL         128
146 #define TASK_WAKING           256
147 #define TASK_PARKED       512 在 get_task_state()返回 TASK_INTERRUPTIBLE
148 #define TASK_STATE_MAX   1024
```

1．TASK_RUNNING

可执行状态，包括就绪和正在 CPU 上执行。只有在该状态的进程才可能在 CPU 上运行。多核平台上同一时刻可能有多个进程处于可执行状态，这些进程的 task_struct 结构（进程控制块）被挂入对应 CPU 的可执行队列（运行队列、就绪队列）中，一个进程最多只能出现在一个 CPU 的运行队列中。很多操作系统教科书将正在 CPU 上执行的进程定义为

RUNNING 状态，而将可执行但是尚未被调度执行的进程定义为 READY 状态，这两种状态在 Linux 下统一为 TASK_RUNNING 状态，对应的状态编码数值为 0。用 ps 命令或/proc/PID/status 查看进程时，可执行状态的进程显示为 R。

2．TASK_INTERRUPTIBLE

可中断的睡眠状态，也是操作系统课程中所谓的阻塞状态。处于这个状态的进程因为等待某事件的发生（通常是 IO 操作，比如等待 socket 连接、等待信号量）而被阻塞。这些进程的 task_struct 结构从运行队列中取下并放入对应事件的等待队列中。当所等待的事件发生时（由外部中断触发或由其他进程触发），对应的等待队列中的一个或多个进程将被唤醒，重新挂回到就绪队列中。通过 ps 命令会看到，除非机器的负载很高，一般情况下进程列表中的绝大多数进程都处于 TASK_INTERRUPTIBLE 状态（ps 相应的输出显示为 S）。

3．TASK_UNINTERRUPTIBLE

不可中断的睡眠状态。与 TASK_INTERRUPTIBLE 状态类似，进程处于阻塞状态，但是此刻进程不会因信号的到来而唤醒。绝大多数情况下，进程处在睡眠状态时，总是应该能够响应异步信号的。由于不响应信号，所以即使用 kill -9 命令也杀不死这样的进程了。这时用 ps 命令或/proc/PID/status 查看进程状态时显示为 D 状态。

而 TASK_UNINTERRUPTIBLE 状态存在的意义就在于，内核的某些处理流程是不能被打断的。如果响应异步信号，程序的执行流程中就会被插入一段用于处理异步信号的流程，于是原有的流程就被中断了。在进程对某些硬件进行操作时（比如进程调用 read 系统调用对某个设备文件进行读操作最终执行到相应设备驱动的代码，并与对应的物理设备进行交互），可能需要使用 TASK_UNINTERRUPTIBLE 状态对进程进行保护，以避免进程与设备交互的过程被打断，造成设备陷入不可控的状态。该状态的进程只能用 wake_up()函数唤醒（例如，驱动程序的中断处理代码中发出调用）。这种情况下的 TASK_UNINTERRUPTIBLE 状态总是非常短暂的，通过 ps 命令基本上不可能捕捉到。

Linux 系统中也存在容易捕捉的 TASK_UNINTERRUPTIBLE 状态。执行 vfork 系统调用后，父进程将进入 TASK_UNINTERRUPTIBLE 状态，直到子进程调用 exit()或 execve()。

4．TASK_STOPPED 或 TASK_TRACED

暂停状态或跟踪状态。向进程发送一个 SIGSTOP 信号，它就会因响应该信号而进入 TASK_STOPPED 状态（除非该进程本身处于 TASK_UNINTERRUPTIBLE 状态而不响应信号）。SIGSTOP 与 SIGKILL 信号一样是强制的，不允许用户进程通过 signal 系列的系统调用重新设置相应的信号处理函数。向进程发送一个 SIGCONT 信号，可以让其从 TASK_STOPPED 状态恢复到 TASK_RUNNING 状态。用 ps 命令或/proc/PID/status 查看这类进程显示的是 T 状态。后台进程执行 getchar()等阻塞操作时会进入 T 状态。

当进程正在被跟踪时，它处于 TASK_TRACED 这个特殊的状态。“正在被跟踪”指的是进程暂停下来，等待跟踪它的进程对它进行操作。比如在 gdb 中对被跟踪的进程设置一个断点，进程在断点处停下来的时候就处于 TASK_TRACED 状态。而在其他时候，被跟踪的进程还是处于前面提到的那些状态。

对于进程本身来说，TASK_STOPPED 和 TASK_TRACED 状态很类似，都是表示进程

暂停下来。而 TASK_TRACED 状态相当于在 TASK_STOPPED 之上多了一层保护，处于 TASK_TRACED 状态的进程不能响应 SIGCONT 信号而被唤醒。只能等到调试进程通过 ptrace 系统调用执行 PTRACE_CONT、PTRACE_DETACH 等操作（通过 ptrace 系统调用的参数指定操作），或调试进程退出，被调试的进程才能恢复 TASK_RUNNING 状态。

5．TASK_DEAD（-EXIT_ZOMBIE）

退出状态，且成为僵尸进程。进程在退出的过程中，处于 TASK_DEAD 状态。在这个退出过程中，进程占有的所有资源将被回收，除了 task_struct 结构（以及少数资源）以外。于是进程就只剩下 task_struct 这么个空壳，故称为僵尸。之所以保留 task_struct，是因为 task_struct 里面保存了进程的退出码，以及一些统计信息。而其父进程很可能会关心这些信息。比如在 shell 中，$?变量就保存了最后一个退出的前台进程的退出码，而这个退出码往往被作为 if 语句的判断条件。保留完整的 task_struct 结构而不仅仅是退出状态，因为在内核中已经建立了从 pid 到 task_struct 的查找关系，还有进程间的父子关系，便于父进程查找。

父进程通过 wait 系列的系统调用（如 wait4、waitid）来等待某个或某些子进程的退出，并获取它的退出信息。然后父进程的 wait 系列系统调用会将子进程的尸体（task_struct）也释放掉。子进程在退出的过程中，内核会给其父进程发送一个信号，通知父进程来“收尸”。这个信号默认是 SIGCHLD，但是在通过 clone 系统调用创建子进程时，可以设置这个信号。

只要父进程不退出且没有对已结束的子进程执行 wait 系统调用，这个子进程就处于僵尸状态并一直持有 task_struct。但是如果父进程结束运行，会将它的所有子进程都托管给别的进程（使之成为别的进程的子进程）——可以是退出进程所在进程组的下一个进程（如果存在的话）或者是 1 号 init 进程，由 init 进程消灭僵尸进程。

用 ps 命令或/proc/PID/status 查看这些进程的时候显示的是 Z 状态。

6．TASK_DEAD（-EXIT_DEAD）

退出状态，且进程即将被销毁。进程在退出过程中也可能不会保留它的 task_struct，比如这个进程是多线程程序中被 detach 过的线程。或者父进程通过设置 SIGCHLD 信号的 handler 为 SIG_IGN 显式地忽略了 SIGCHLD 信号，子进程结束后将被置于 EXIT_DEAD 退出状态，这意味着接下来的内核代码立即就会将该进程彻底释放。所以 EXIT_DEAD 状态是非常短暂的，几乎不可能通过 ps 命令捕捉到（显示为 X 状态）。

5.2.2 进程状态变迁

刚创建的时候处于可执行就绪状态，然后根据条件的不同在各个状态之间变化，直到退出。

1．进程初始状态

进程是通过 fork 系列的系统调用（fork、clone、vfork）来创建的，内核（或内核模块）也可以通过 kernel_thread()函数创建内核进程。那么既然调用进程处于 TASK_RUNNING 状态，则子进程自然也处于 TASK_RUNNING 状态。另外，在 clone 系统调用和内核函数

kernel_thread()中也接受 CLONE_STOPPED 选项，从而将子进程的初始状态置为 TASK_STOPPED。

2．状态转换

进程状态转换有好几种，但是进程状态的变迁主要方向却只有两个——从 TASK_RUNNING 状态变为非 TASK_RUNNING 状态，或者从非 TASK_RUNNING 状态变为 TASK_RUNNING 状态。图 5-2 的中间是 TASK_RUNNING 运行状态——具体再细分为是否占有 CPU，如果占有 CPU 又分为用户态和内核态。两边则是非 TASK_RUNNING 状态。

从 RUNNING 状态转入阻塞睡眠有两种情况，一种是不可中断睡眠，另一种是可中断睡眠。而从阻塞睡眠状态到 RUNNING 状态则需要调用 wake_up()来实现。

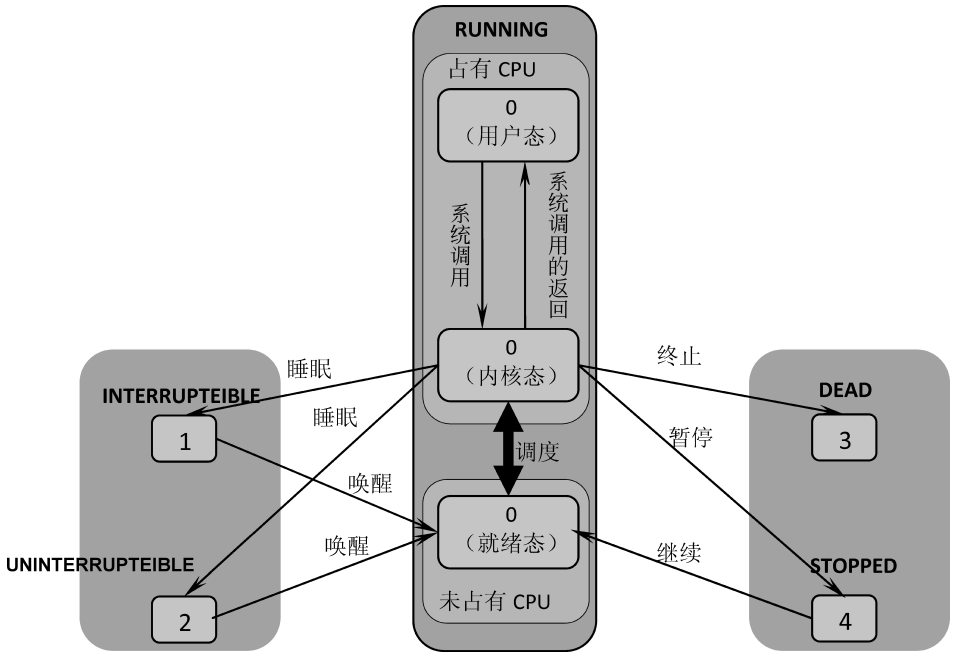

图 5-2　进程状态转换图

操作系统概念上的调度包含图 5-2 中的所有状态转换，但是本书中 5.3 节调度算法讨论的是图 5-2 中关于如何从就绪进程中选择一个进程来使用 CPU 的部分（即图中粗线条的双向箭头）。

可以看出，如果给一个 TASK_INTERRUPTIBLE 状态的进程发送 SIGKILL 信号，这个进程将先被唤醒（进入 TASK_RUNNING 状态），然后再响应 SIGKILL 信号而退出（变为 TASK_DEAD 状态）。并不会从 TASK_INTERRUPTIBLE 状态直接退出。

进程从非 TASK_RUNNING 状态变为 TASK_RUNNING 状态，是由别的进程（也可能是中断处理程序）执行唤醒操作来实现的。执行唤醒的进程设置被唤醒进程的状态为 TASK_RUNNING，然后将其 task_struct 结构加入到某个 CPU 的运行队列中。于是被唤醒的进程将有机会被调度执行。

而进程从 TASK_RUNNING 状态变为非 TASK_RUNNING 状态，则有几种途径：①响应信号而进入 TASK_STOPED 状态或 TASK_DEAD 状态；②执行系统调用主动进入

TASK_INTERRUPTIBLE 状态（如 nanosleep 系统调用）或 TASK_DEAD 状态（如 exit 系统调用）；③由于执行系统调用需要的资源得不到满足，而进入 TASK_INTERRUPTIBLE 状态或 TASK_UNINTERRUPTIBLE 状态（如 select 系统调用）。显然，这些情况都只能发生在进程正在 CPU 上执行的情况下。

请注意图 5-1 中各个 CPU 的运行队列（rq 数据结构所管理）的进程仅仅是图 5-2 中就绪的、未占有 CPU 的那部分 RUNNING 进程，而各 CPU 上正在运行的进程由各自的 current 宏所指向，未能执行的阻塞状态进程（无论是否可中断）则在各自的等待队列中。没有全局统一的 INTERRUPTIBLE 或 UNINTERRUPTIBLE 等队列，而是分散在系统各处。例如，进行消息队列通信的阻塞进程按照发送和接收操作的不同，分别将各自的 task_struct 包含在消息队列的 msg_receiver 和 msg_sender 结构体内，再插入到相应的队列，具体见图 6-12。

5.3 进 程 调 度

从图 5-2 可知，调度工作是将就绪的任务按一定准则排序，逐个使用 CPU 的过程。Linux 调度和负载均衡是有一定耦合的，本节只讨论调度，负载均衡在 5.4 节讨论。

调度的目的是为了在进程间共享处理器等硬件资源，实现公平高效的执行。而公平高效并没有一个统一的标准，因此任何一个操作系统的调度实现都必须在确定的调度目标上实现。调度工作涉及两个要素，一个是就绪队列如何组织管理，另一个就是何时以及根据什么准则从就绪队列中选择一个进程来执行。不同属性的任务可能需要不同的选择准则。

关于调度需要弄清楚：①调度框架和调度函数的工作流程；②调度算法；③调度时机；④进程切换操作。

5.3.1 调度框架

Linux 的调度器提供了一个基本的框架，可以容纳不同的调度准则和算法。默认情况下它首先考虑任务的紧迫程度差异性——分为实时调度类任务和普通调度类的任务，先调度实时任务再调度普通任务。实时任务可以有两类调度算法：FIFO 和 RR；普通任务则使用完全公平调度算法 CFS。

进程描述符中与调度相关的一些成员如代码 5-2 所示，后面讨论会逐个分析。

代码 5-2 task_struct 中的部分调度信息 (linux-3.13/include/linux/sched.h)

```
1042 struct task_struct {
...
1058        int on_rq;
1059
1060        int prio, static_prio, normal_prio; 参见 5.3.1 节的“优先级”小节
1061        unsigned int rt_priority;
1062        const struct sched_class *sched_class; 本进程的调度类(含紧迫程度信息)
1063        struct sched_entity se;                参见 5.3.2 节的 CFS 调度实体
```

```
1064        struct sched_rt_entity rt;              参见 5.3.3 节的 RT 调度实体
...
1078        unsigned int policy;                    调度策略
1079        int nr_cpus_allowed;                    CPU 调度\绑定的处理器个数
1080        cpumask_t cpus_allowed;                 CPU 调度\绑定的处理器位图
```

所有进程都根据所属的调度类和调度算法，组织归并到各个 CPU 上相应类型的运行队列之上。图 5-3 上半部分显示了 4 个 CPU，各 CPU 上的运行队列 rq 分别管理着 RT 实时进程和 CFS 普通进程，以及 stop 和 idle 指向的两个特殊进程（stop 指针通常为空，idle 总是指向 idle 进程）。各 CPU 上同一个队列上的所有进程属于同一种调度器类，即它们的 task_struct->sched_class 指向同一个调度器类实例。

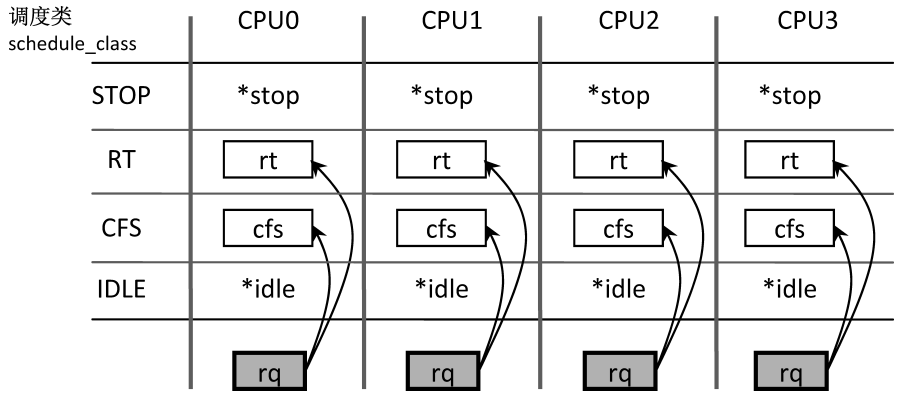

图 5-3　调度队列、负载均衡域与 CPU 的对应关系

1．调度队列

如图 5-3 所示，所有就绪（TASK_RUNNING）的任务被安排在各个处理器私有的就绪队列（运行队列）中，各个处理器从自己的就绪队列中选择任务来执行。每 CPU 变量 rq（runqueues 结构体）管理着每个处理器上的就绪任务，它声明如下（kernel/sched/sched.h）：

```
545 DECLARE_PER_CPU(struct rq, runqueues);
```

我们以双核处理器（P0/P1）为例说明这些队列的组织关系，普通进程和实时进程分别组织管理，具体如图 5-4 所示。普通进程实际上是按照红黑树管理，使用队列这个称呼完全是历史原因。

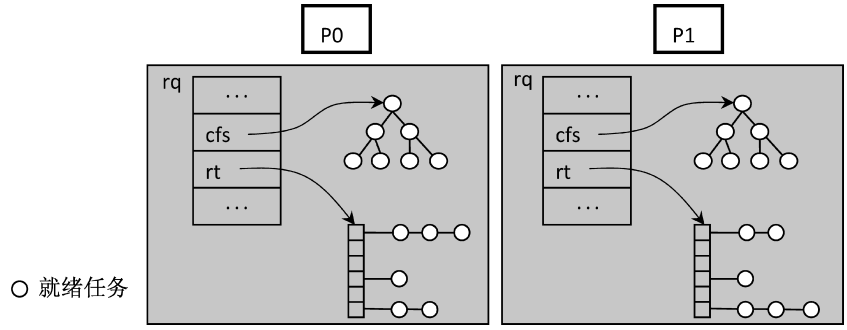

图 5-4　双核处理器上的运行队列示意图

每个处理器的 rq 主要管理了两种不同性质的任务队列，第一类由 rq->rt 指向实时任务，第二类由 rq->cfs 指向普通任务。另有两个 task_struct 结构体指针 rq->idle 和 rq->stop，分别指向处理器空闲时运行的进程和停止处理器时运行的进程。运行队列 rq 的定义具体见代码 5-3。

代码 5-3　rq（linux-3.13/kernel/sched/sched.h）

```
403 struct rq {
404         /* runqueue lock: */
405         raw_spinlock_t lock;
406
407         /*
408          * nr_running and cpu_load should be in the same cacheline because
409          * remote CPUs use both these fields when doing load calculation.
410          */
411         unsigned int nr_running;                    就绪任务数
412 #ifdef CONFIG_NUMA_BALANCING
413         unsigned int nr_numa_running;
414         unsigned int nr_preferred_running;
415 #endif
416         #define CPU_LOAD_IDX_MAX 5
417         unsigned long cpu_load[CPU_LOAD_IDX_MAX];
418         unsigned long last_load_update_tick;
419 #ifdef CONFIG_NO_HZ_COMMON
420         u64 nohz_stamp;
421         unsigned long nohz_flags;
422 #endif
423 #ifdef CONFIG_NO_HZ_FULL
424         unsigned long last_sched_tick;
425 #endif
426         int skip_clock_update;
427
428         /* capture load from *all* tasks on this cpu: */
429         struct load_weight load;          本队列所有进程的总权重，用于均衡
430         unsigned long nr_load_updates;    负载更新的次数（每 tick 都增 1）
431         u64 nr_switches;                  已经进行的进程切换次数
432
433         struct cfs_rq cfs;                普通进程的公平调度队列，见 5.3.2 节
434         struct rt_rq rt;                  实时进程的调度队列，见 5.3.3 节
435
436 #ifdef CONFIG_FAIR_GROUP_SCHED
437         /* list of leaf cfs_rq on this cpu: */
438         struct list_head leaf_cfs_rq_list;
439 #endif /* CONFIG_FAIR_GROUP_SCHED */
440
441 #ifdef CONFIG_RT_GROUP_SCHED
442         struct list_head leaf_rt_rq_list;
```

```
443 #endif
444
445         /*
446          * This is part of a global counter where only the total sum
447          * over all CPUs matters. A task can increase this counter on
448          * one CPU and if it got migrated afterwards it may decrease
449          * it on another CPU. Always updated under the runqueue lock:
450          */
451         unsigned long nr_uninterruptible;
452
453         struct task_struct *curr, *idle, *stop; idle和stop分别对应IDLE
和STOP调度类任务
454         unsigned long next_balance;                下一次执行负载均衡的时间
455         struct mm_struct *prev_mm;
456
457         u64 clock;
458         u64 clock_task;
459
460         atomic_t nr_iowait;
461
462 #ifdef CONFIG_SMP
463         struct root_domain *rd;
464         struct sched_domain *sd;
465
466         unsigned long cpu_power;
467
468         unsigned char idle_balance;  置1表示本处理器空闲，需要负载均衡
469         /* For active balancing */
470         int post_schedule;
471         int active_balance;
472         int push_cpu;
473         struct cpu_stop_work active_balance_work;
474         /* cpu of this runqueue: */
475         int cpu;                        本rq所属的CPU
476         int online;
477
478         struct list_head cfs_tasks;
479
480         u64 rt_avg;
481         u64 age_stamp;
482         u64 idle_stamp;
483         u64 avg_idle;                   平均空闲期，用于判断是否进行idle均衡
484
485         /* This is used to determine avg_idle's max value */
486         u64 max_idle_balance_cost;
487 #endif
488
```

```
489 #ifdef CONFIG_IRQ_TIME_ACCOUNTING
490         u64 prev_irq_time;
491 #endif
492 #ifdef CONFIG_PARAVIRT
493         u64 prev_steal_time;
494 #endif
495 #ifdef CONFIG_PARAVIRT_TIME_ACCOUNTING
496         u64 prev_steal_time_rq;
497 #endif
498
499         /* calc_load related fields */
500         unsigned long calc_load_update;
501         long calc_load_active;
502
503 #ifdef CONFIG_SCHED_HRTICK
504 #ifdef CONFIG_SMP
505         int hrtick_csd_pending;
506         struct call_single_data hrtick_csd;
507 #endif
508         struct hrtimer hrtick_timer;
509 #endif
510
511 #ifdef CONFIG_SCHEDSTATS
512         /* latency stats */
513         struct sched_info rq_sched_info;
514         unsigned long long rq_cpu_time;
515         /* could above be rq->cfs_rq.exec_clock + rq->rt_rq.rt_runtime ? */
516
517         /* sys_sched_yield() stats */
518         unsigned int yld_count;
519
520         /* schedule() stats */
521         unsigned int sched_count;
522         unsigned int sched_goidle;
523
524         /* try_to_wake_up() stats */
525         unsigned int ttwu_count;
526         unsigned int ttwu_local;
527 #endif
528
529 #ifdef CONFIG_SMP
530         struct llist_head wake_list;
531 #endif
532
533         struct sched_avg avg;
534 };
```