操作系统课程作业报告

——Linux Kernel 2.6 进程调度的分析与改进

学校：浙江大学

学院：计算机学院

姓名：（20321131）

目录

第一章 Kernel 2.4 进程调度回顾.....3
一、Kernel 2.4 进程调度简单介绍.....3
二、Kernel 2.4 存在的不足.....3
第二章 Kernel 2.6 进程调度分析.....3
一、基本思想.....3
1. 就绪队列的改进.....3
2. 快速查找应该执行的进程.....4
3. 引进 “ load estimator”.....4
4. 内核可抢占.....4
5. 调度器相关的负载均衡.....4
二、数据结构.....4
1. 进程优先级的划分.....4
2. 就绪队列 runqueue（kernel/sched.c）.....4
3. 进程标识 task_struct(include/linux/sched.h).....6
三、调度策略.....9
1. 进程优先级.....9
2. 进程时间片.....9
3. 平均等待时间 sleep_avg.....10
4. 交互进程优化.....11
5. 调度器主函数 schedule()（kernel/sched.c）.....13
6. 进程调度的生与死.....15
7. 内核可抢占.....17
8. 负载均衡.....18
第三章 对 Kernel 2.6 的一点改进.....20

第一章 Kernel 2.4 进程调度回顾

一、Kernel 2.4 进程调度简单介绍

略

二、Kernel 2.4 存在的不足

根据以上对 2.4 进程调度的分析，我们总结出看出 2.4 内核总的特点就是：

- 内核调度简单有效
- 内核不可抢占

但是经过对 2.4 内核的分析，我们也明显看到了它的缺点：

1. 调度算法复杂度是 O(n)，与系统负荷关系较大。而且调度算法在设计上也有缺陷，比如：

(1) 2.4 进程调度只设置了一个进程就绪队列，这样有的进程用完了自己时间片以后还要呆在就绪进程队列里面。这样这个进程虽然在这一轮调度循环里面已经无法取得 CPU 的使用权，但是还要参与 goodness()值的计算，这样就白白浪费了时间。

(2) 就绪进程队列是一个全局数据结构，多个 CPU 只有一个就绪队列 runqueue，因而调度器对它的所有操作都会因全局自旋锁而导致系统各个处理机之间的等待，使得就绪队列成为一个明显的瓶颈。

2. 调度算法在内核态不可抢占。如果某个进程一旦进了内核态那么再高优先级的进程都无法剥夺，只有等进程返回内核态的时候才可以进行调度。
3. 缺乏对实时进程的支持。

第二章 Kernel 2.6 进程调度分析

以下分析都基于 Linux Kernel 2.6.4。

一、基本思想

Kernel2.6 调度算法仍然是基于优先级的调度，它的算法复杂度为 O(1)，也就是说是调度器的开销是恒定的，与系统当前的负载没有关系。

1. 就绪队列的改进

每个 CPU 有两个按优先级排序的数组：一个是 active array；一个是 expired array。

Active array 是当前 CPU 可能选择执行的运行进程队列，队列中的每个进程都有时间片剩下。Expired array 是那些用户时间片的就绪进程队列。一旦 active array 里面某个普通进程的时间片用完了，调度器将重新计算进程的时间片、优先级，将它从 active array 中删除，插入到 expired array 中相应得优先级队列中。Active array 和 expired array 是通过两个指向每个 CPU 运行队列的指针来访问的。所以当 active array 中所有的进程都用完时间片，只需将两个指针切换一下就可以了，这比 Kernel 2.4 的切换要改进了很多。

2. 快速查找应该执行的进程

系统中往往有很多的就绪进程，如何快速找到 CPU 即将运行的进程就成了关系到系统性能的一个重要因素。针对 2.4 的缺点，Kernel 2.6 进行了重新设计：

引进了一个 64bit 的 bitmap 作为进程队列的索引，用 bitmap 来记载某个优先级的进程队列上有无进程，如果有则为 1。这样使得寻找优先级最高的任务只需要两个 BSFL 命令。

3. 引进“ load estimator ”

在一个负载很重的系统上有一个很好的交互感是一件很困难的事情，设计者经过研究发现一味的激励（boost）交互任务并不够，还需惩罚（punish）那些需求大于可获得 CPU 时间的进程。调度器通过对用户睡眠时间和运行时间的纪录来判断进程是否是交互进程，一旦被认为是交互进程，调度器会给进程很多“ 奖励 ”（bonus）。

4. 内核可抢占

内核可抢占可以说是 2.6 内核调度器优于 2.4 内核的一个很重要的原因。

当内核进程没有访问内核的关键数据，也就是内核没有被加锁，此时内核代码是可重入的，因此更高优先级的进程可以在此时中断正在执行的进程，从而达到抢占的目的。

5. 调度器相关的负载均衡

负载均衡有两种策略，一种是从别的 CPU 上将进程迁移过来，称为“ pull ”；一种是将本 CPU 上的进程迁移出去，称为“ push ”。

二、数据结构

1. 进程优先级的划分

Kernel 2.6 将进程优先级作了以下规定：

进程优先级范围是从 0 ~ MAX_PRIO-1，其中实时进程的优先级的范围是 0 ~ MAX_RT_PRIO-1，普通进程的优先级是 MAX_RT_PRIO ~ MAX_PRIO-1。数值越小优先级越高。

2. 就绪队列 runqueue（kernel/sched.c）

struct runqueue 是 2.6 调度器中一个非常重要的数据结构，它主要用于存放每个 CPU 的就绪队列信息。限于篇幅，这里只介绍其中相对重要的部分：

(1) prio_array_t *active, *expired, arrays[2]

这是 runqueue 中最重要的部分。每个 CPU 的就绪队列都是一个数组，按照时间片是否用完将就绪队列分为两个部分，分别用指针 active 和 expired 来指向数组的两个下标。prio_array_t 的结构如下：

```
struct prio_array {
    int nr_active;                          /*本进程组中进程个数*/
    struct list_head queue[MAX_PRIO];       /*每个优先级的进程队列*/
    unsigned long bitmap[BITMAP_SIZE];      /*上述进程队列的索引位图*/
```

```
};
```

程序 1　prio_array 结构体

数组 queue[MAX_PRIO]里面存放的是优先级为 i（MAX_PRIO>i>=0）的进程队列的链表头，即 task_struct::runlist（通过 runnlist 即可找到 task_struct）。它的具体结构为：

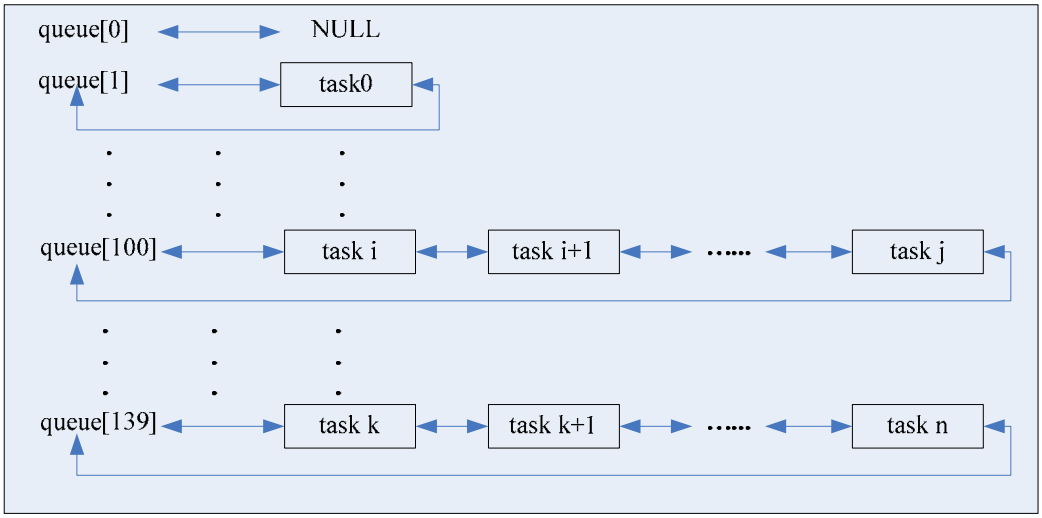

图 1　queue[MAX_PRIO]队列

那么调度器在执行调度的任务时是怎么找到优先级最高的进程呢？

在结构体 struct prio_array 中有一个重要的数据 unsigned long bitmap[BITMAP_SIZE]，这个数据是用来作为进程队列 queue[MAX_PRIO]的索引位图，bitmap 的每一位（bit）都与 queue[i]对应。当 queue[i]的进程队列不为空时，bitmap 的相应位就为 1；否则就为 0。这样我们只需要通过汇编指令从进程优先级由高到低的方向找到第一个为 1 的位置 idx 即为当前就绪队列中最高的优先级（函数 sched_find_first_bit()就是用来完成这一工作的），那么 queue[i]->next 就是我们要找的 task_struct::runlist。

当一个普通进程的时间片用完以后将重新计算进程的时间片和优先级，将该进程从 active array 中删除，添加到 expired array 中相应优先级的进程队列中。当 active array 中没有进程时，则将 active 和 expired 指针调换一下就完成了切换工作。而在 2.4 内核中重新计算时间片是在所有就绪进程的时间片都用完以后才统一进行的，因而进程时间片的计算非常耗时，而在 2.6 中计算时间片是分散的，而且通过以上的方法来实现时间片的轮转，这也是 2.6 调度器一个亮点。

另外，程序将 struct runqueue 定义在 sched.c 里面而没有定义在 sched.h 里面是为了让抽象调度器部分的代码，使得内核的其他部分使用调度器提供的接口即可。

(2) spinlock_t lock

runqueue 的自旋锁，当对 runqueue 进行操作的时候，需要对其加锁。由于每个 CPU 都有一个 runqueue，这样会大大减少竞争的机会。

(3) task_t *curr

CPU 当前运行的进程。

在程序中还有一个全局变量 current 也是 CPU 当前运行的进程，它在通常情况下和 runqueue 的 curr 指针是相同的，但是当调度器进行调度的时，如果已经找到最高优先级的进程，则此时做 rq->curr = next;可见在进行任务切换之前，rq->curr 和 current 的值是不同的。当唤醒一个进程的时候，很明显将唤醒进程与 rq->curr 的优先级进行比较更有意义。

(4) unsigned long expired_timestamp

此变量是用来记录 active array 中最早用完时间片的时间（赋值 jiffies）。因此，用这个量就可以记录 expired array 中等时间最长的进程的等待时间。这个值的主要用处是用于宏 EXPIRED_STARVING()（这个宏主要是用来判断 expired array 中的进程是否已经等待了足够长的时间，详见“进程调度的生与死”一节中“scheduler_tick()”函数的介绍）。

(5) unsigned long nr_running, nr_switches, nr_uninterruptible, timestamp_last_tick

用来记录该 CPU 进程相关数据。具体作用如下

- ✧ nr_running　　记录该 CPU 上就绪进程总数，是 active array 和 expired array 进程总数和
- ✧ nr_switches　　记录该 CPU 运行以来发生的进程切换次数
- ✧ nr_uninterruptible　　记录该 CPU 不可中断状态进程的个数
- ✧ timestamp_last_tick　　记录就绪进程队列上次发生调度的时间，用于负载均衡

(6) struct list_head migration_queue

这个是存放希望迁移到其他 CPU 上的进程队列，实际迁移的数据类型是 migration_req_t，这里是通过将 migration_req_t::list 连接起来。详见“负载均衡”中“push”一节。

3. 进程标识 task_struct(include/linux/sched.h)

Linux 是一个多任务的操作系统，在多任务操作系统中每一个进程都由一个 PCB 程序控制块来标识在 Linux 中 PCB 实际上是一个名为 task_struct 的结构体。

task_struct 有上百个域，主要包括了 10 个方面的信息：1.进程状态；2.调度信息，如调度策略，优先级，时间片，交互值等；3.进程的通讯状况；4.进程树中的父子兄弟的指针；5.时间信息，如睡眠时间，上一次发生调度时间等；6.标号，决定该进程归属；7.打开的一些文件信息；8.进程上下文和内核上下文；9.处理器上下文；10.内存信息。

由于 task_struct 结构体比较复杂，因此我们只注意它与进程调度相关的重要部分。

(1) volatile long state

进程所处的状态。在 include/linux/sched.h 中包含 6 种状态：

- ✧ #define TASK_RUNNING　　0
- ✧ #define TASK_INTERRUPTIBLE　　1
- ✧ #define TASK_UNINTERRUPTIBLE　　2
- ✧ #define TASK_STOPPED　　4
- ✧ #define TASK_ZOMBIE　　8
- ✧ #define TASK_DEAD　　16

新增的 TASK_DEAD 是表示已经退出且不需父进程回收的进程的状态。

(2) struct thread_info *thread_info

当前进程运行的一些环境信息。其中有两个结构成员非常重要，与调度密切相关：

- ✧ __s32　　preempt_count;
- ✧ unsigned long　　flags;

preempt_count 是用来表示内核能否被抢占的使能成员。如果它大于 0，表示内核不能被抢占；如果等于 0，则表示内核处于安全状态（即没有加锁），可以抢占。

flags 里面有一个 TIF_NEED_RESCHED 位，它和 Kernel 2.4 中 need_resched 作用一样。如果此标志位为 1，则表示应该尽快启动调度器。

(3) int prio, static_prio

prio 是进程的动态优先级，相当于 Kernel2.4 中用 goodness()函数计算出来的结果；在 Kernel2.6 中不再是由调度器统一计算，而是独立计算；prio 的计算和许多因素有关，详见“进程优先级的计算”一节。

static_prio 则是进程的静态优先级，与 nice 意义相同。nice 的取值仍然是-20 ~ 19，数值越小，进程优先级越高。

kernel/sched.c 中定义了两个宏来完成将 nice 转换到 prio 的取值区间和将 prioity 转换到 nice 取值区间。

- #define NICE_TO_PRIO(nice) (MAX_RT_PRIO + (nice) + 20)
- #define PRIO_TO_NICE(prio) ((prio) - MAX_RT_PRIO - 20)

可见 prioity 和 nice 的关系是：

$$priority = MAX\_RT\_PRIO + nice + 20$$

(4) struct list_head run_list

前面提到过，就绪进程都是按照优先级进行排列，prio_array 中的 queue[MAX_PRIO]存放的是指向每个优先级队列的链头 list_head；而同一优先级的进程则是通过 run_list 链接在一起。

include/linux/list.h 定义了一种抽象的双向链表 struct list_head，通过它可以将任意类型的结构体链接到一起。task_struct 也是通过这种方式链接起来的。

(5) prio_array_t *array

指向当前 CPU 的 active array 的指针。在进程控制块里面又加了一个指向 active array 的指针，看似重复，其实不然。比如说对于下面的代码（kernel/sched.c）：

```
array = next->array;
dequeue_task(next, array);
recalc_task_prio(next, next->timestamp + delta);
enqueue_task(next, array);
```

程序 2　示例代码

对于单处理器（UP）的情况，我们确实可以通过 runqueue::active 直接得到当前的 active array；但是对于 SMP，就不是这样了，需要引用 next 的 thread_info，再依靠 thread_info 中的 cpu 找到 next 所在的处理器，找到以后再找到这个 cpu 上的 runqueue，最后得到 active。对于 schedule 这样频繁调用的函数，这种浪费是不能容忍的。

(6) unsigned long sleep_avg

进程的平均等待时间，单位是纳秒（nanosecond），在 0 ~ NS_MAX_SLEEP_AVG 范围内。它的实质是进程等待时间和运行时间的差值。当进程处于等待或者睡眠状态时，该值变大；当进程运行时，该值变小。

sleep_avg 是 Kernel 2.6 中衡量进程的一个关键指标，它既可以用来衡量进程的交互程度，也可以用来衡量进程的紧急程度。具体内容将在“平均等待时间 sleep_avg”一节作详细介绍。

(7) long interactive_credit

表示进程交互程度，取值范围在-CREDIT_LIMIT ~ CREDIT_LIMIT+1 之间。进程创建的时候值为 1，以后根据不同的情况进行不同的增 1、减 1；如果一个进程的 interactive_credit 超过 CREDIT_LIMIT 之后，这个进程就会被认为是交互式进程，同时 interactive_credit 的值

也就不再改变了（恒为 CREDIT_LIMIT+1）。下面将在“ 交互进程优化 ”一节详细介绍。

(8) unsigned long long timestamp

进程发生调度的时间，单位和 sleep_avg 一样，也是纳秒。它负责纪录以下四种情况的时间：

- ✧ 进程被唤醒的时间：

  在 activate_task()（kernel/sched.c）中记录（p->timestamp = now）。
- ✧ 进程被切换到 expired array 的时间：

  在 schedule()（kernel/sched.c）中记录，当准备进行进程切换的时候，记录下该进程被切换到 expired array 的时间（prev->timestamp = now）。
- ✧ 进程被切换到 active array 的时间：

  在 schedule()（kernel/sched.c）中记录，进行进程切换的开始，记录下下一个进程被切换到 active array 的时间（next->timestamp = now）。
- ✧ 负载均衡相关的赋值

  在进行负载均衡的时候，当把一个进程从其他 CPU 上 pull 过来的时候需要将该进程的 timestamp 设成 sched_clock() - (src_rq->timestamp_last_tick - p->timestamp)，即相对于本 CPU 被切换下来的时间。

(9) int activated

表示该进程被唤醒的类别：

- ✧ actived=-1 表示该进程并非自愿 sleep，其先前状态是 TASK_UNINTERRUPTIBLE。在 try_to_wake_up()中设置。
- ✧ actived=0 缺省值，表示进程本来就是处于就绪状态。
- ✧ actived=1 进程先前状态是 TASK_INTERRUPTIBLE，但是不是由中断唤醒；这样的进程在第一次运行时有 credit，以后就没有了。在 activate_task()中设置。
- ✧ actived=2 进程先前状态是 TASK_INTERRUPTIBLE，进程被中断唤醒。这样的进程非常像交互式进程。在 activate_task()中设置。

(10) unsigned long policy

进程的调度策略和 2.4 一样，有以下几种：

- ✧ SCHED_FIF 先进先出式调度，除非有更高优先级进程申请运行，否则该进程将保持运行至退出才让出 CPU
- ✧ SCHED_RR 轮转式调度，该进程被调度下来后将被置于运行队列的末尾，以保证其他实时进程有机会运行）
- ✧ SCHED_OTHER 常规的分时调度策略

(11) unsigned int time_slice, first_time_slice

time_slice 是进程剩余的时间片，相当于 Kernel 2.4 里面 counter，但是时间片不再影响进程的优先级。

first_time_slice 用来记录时间片是否是第一次分配（进程创建时），如果值不为 0，进程退出时将时间片交还给父进程。

三、调度策略

1. 进程优先级

(1) 优先级的计算

前面已经说过，优先级由两部分构成，一是静态优先级 static_prio，一是动态优先级 prio。静态优先级在进程创建的时候就被赋值，并且不变（除非用系统调用改变进程的 nice 值）；而进程的动态优先级则是跟 static_prio 和 sleep_avg 有关。对于实时进程的优先级在创建的时候就确定了，而且一旦确定以后就不再改变，所以下面部分仅对于非实时进程而言。具体的计算由函数 effecitve_prio()（kernel/sched.c）完成。

函数将进程的 sleep_avg 映射成范围是-MAX_BONUS/2 ~ MAX_BONUS/2 的变量 bonus，而 MAX_BONUS 是等于 MAX_USER_PRIO * PRIO_BONUS_RATIO / 100 = 10，可见 sleep_avg 仅能影响的优先级范围在-5 ~ 5 之间。具体的映射是由以下规则完成的：

$$bonus = (\text{NS\_TO\_JIFFIES}((p)->\text{sleep\_avg}) * \text{MAX\_BONUS} / \text{MAX\_SLEEP\_AVG}) - \text{MAX\_BONUS} / 2$$

那么进程的动态优先级就等于：$prio = static\_prio - bonus$（当然必须在 MAX_RT_PRIO 和 MAX_PRIO-1 之间）。可见，sleep_avg 和 bonus 是一个线性关系。进程的 sleep_avg 越大，bonus 越大，从而进程的动态优先级也就越高。

(2) 何时计算优先级

计算进程的动态优先级一般调用两个函数，一个是 effective_prio()，一个是 recalc_task_prio ()。函数 recalc_task_prio ()先要根据进程被唤醒前的状态（即 actived）、interactive_credit 等来计算进程的 sleep_avg（详见“平均等待时间 sleep_avg”一节），在最后调用 effective_prio()来计算函数的动态优先级。

总的来说，有以下几种情况需要计算进程的优先级：

- ✧ a. 创建新进程，使用函数 effective_prio()（因为此时进程尚未进行调度，没有 sleep_avg 和 interactive_credit 可言）；
- ✧ b. 唤醒等待进程时，使用函数 recalc_task_prio ()来计算进程动态优先级。
- ✧ c. 进程用完时间片以后，被重新插入到 active array 或者 expired array 的时候需要重新计算动态优先级，以便将进程插入到队列的相应位置。此时，使用函数 effective_prio()；
- ✧ d. 其他情况，如 IDLE 进程初始化等时候。

2. 进程时间片

(1) 时间片的计算

进程的时间片 time_slice 是基于进程静态优先级的，静态优先级越高（值越小），时间片就越大。计算时间片是同过函数 task_timeslice()（kernel/sched.c）来完成的。该函数也是使用线性映射的方法，将进程优先级[MAX_RT_PRIO, MAX_PRIO-1]映射到时间片[MIN_TIMESLICE, MAX_TIMESLICE]范围内。通过优先级来计算时间片的等式为：

```
timeslice = MIN_TIMESLICE + ((MAX_TIMESLICE - MIN_TIMESLICE) *
            (MAX_PRIO-1- (p)->static_prio) / (MAX_USER_PRIO-1))
```

程序　时间片的计算公式

可以用下面的图 2 来形象的表示 static_prio 和 timeslice 之间的关系：

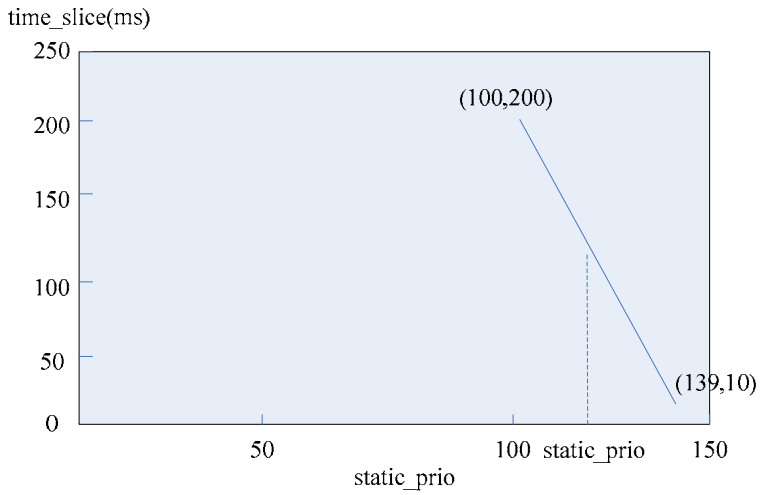

图 2　static_prio 和 time_slice 之间的关系

(2) 何时计算时间片

当就绪进程的所有进程的时间片都是 0 的时候，许多操作系统（包括旧版本的 Linux）是使用下面的循环来给进程队列计算时间片的：

```
for (each task on the system) {
        recalculate priority;
        recalculate timeslice
}
```

程序　一般操作系统计算时间片的方法

这样的循环计算会导致以下问题：

- ✧ 循环可能会花很长时间，而且算法的复杂度 O(n)；
- ✧ 计算过程中必须给进程队列和 task_struct 上锁，这样可能导致大量的竞争；
- ✧ 因为计算时间不可预计，所以可能给实时进程带来问题；

在 Kernel 2.6 中时间片的计算是分散的，具体的计算既可以用 task_timeslice()，也可以用其他方法。

- ✧ a. 进程创建时，将父进程的时间片分一半给子进程，同时父进程的时间片减半。(详见“ sched_fork ”一节)；
- ✧ b. 进程用完时间片以后，需要重新计算时间片，并将进程插入到相应的运行队列。（详见“ scheduler_tick ”一节）；
- ✧ c. 进程退出时，根据 first_timeslice 的值来决定是否将子进程的时间片返还给父进程。(详见“ 退出调度 ”一节)。

可见 Kernel2.6 通过分散计算时间片的办法很好解决了上面循环计算所带来的几个问题。

3. 平均等待时间 sleep_avg

平均等待时间 sleep_avg 既决定了进程优先级，又影响了进程交互程度的，因此它是 Kernel 2.6 调度系统里面很复杂的一块。下面将跟踪调度器中 sleep_avg 的变化情况。

(1) 进程创建

当一个进程被创建的时候，父进程的 sleep_avg 要乘以“ PARENT_PENALTY / 100 ”，子

进程的 sleep_avg 要乘以“ CHILD_PENALTY / 100 ”， PARENT_PENALTY=100，而 CHILD_PENALTY = 95，可见创建以后子进程的 sleep_avg 要降低，而父进程则不变。

(2) 进程被唤醒

当一个进程被唤醒以后，acitvate_task()将调用函数 recalc_task_prio()来计算进程的 sleep_avg，参数是进程的睡眠时间，从而进一步计算进程的动态优先级。计算 sleep_avg 有以下几种可能（当然都需在 0 ~ NS_MAX_SLEEP_AVG 范围内）：

- ✧ MAX_SLEEP_AVG - AVG_TIMESLICE

  当用户进程（p->mm）不是由 UNINTERRUPTIBLE 状态唤醒（p->activated != -1），且睡眠时间大于 INTERACTIVE_SLEEP(p)，则做此赋值；
- ✧ 不变

  当用户进程（p->mm）是由 UNINTERRUPTIBLE 状态唤醒（p->activated == -1），且“ 交互程度 ”不高 （!HIGH_CREDIT(p)），如果原来的 sleep_avg 已经大于 INTERACTIVE_SLEEP(p)，则不变（对非自愿睡眠的进程进行惩罚）； 否则见下面一条；
- ✧ INTERACTIVE_SLEEP(p)

  如果加上此次的睡眠时间后大于 INTERACTIVE_SLEEP(p)，则 sleep_avg 赋值为 INTERACTIVE_SLEEP(p)；
- ✧ sleep_avg+sleep_time

  如果以上条件全都不满足，则直接将本次睡眠时间加到 sleep_avg 上。

(3) 进程调度过程中

在 schedule()过程中，如果发现优先级最高的程序是刚刚从 TASK_INTERRUPTIBLE 状态被唤醒的进程（actived>0，参见“ actived ”的定义），那么将调用 recalc_task_prio()，运算过程与(2)相同，所不同的就是调用时的参数 sleep_time 是进程在就绪队列的等待时间。如果进程不是被中断唤醒的（actived=1），那么 sleep_time 还将受到“ (ON_RUNQUEUE_WEIGHT * 128 / 100) / 128 ”的限制，因为该进程很可能不是交互式进程。

(4) 进程被剥夺 CPU 使用权

当进行进程切换的时候，被剥夺 CPU 使用权的进程的 sleep_avg 将会被减去进程的运行时间 run_time（这里的 run_time 对于交互式进程也有奖励的，详见“ 交互式进程优先 ”一节），从而保证调度器的公平性。进程运行的时间越长，sleep_avg 就越小（底限是 0），进程的动态优先级也就越低，从而被调度器调度到的机会也就会越小。

(5) 进程退出

当一个进程退出时，如果该进程的 sleep_avg 比父进程要小（也就是运行时间长），那么父进程将得到惩罚。具体惩罚的规则为：

```
p->parent->sleep_avg = p->parent->sleep_avg / (EXIT_WEIGHT+1) * EXIT_WEIGHT +
p->sleep_avg /(EXIT_WEIGHT + 1);
```

父进程的 sleep_avg 将变为原来的 1/(EXIT_WEIGHT+1)，再加上子进程的 sleep_avg 的 1/(EXIT_WEIGHT+1)，可见子进程运行的越多，父进程得到的惩罚也就越大。这样也是为了保证调度器的公正性。

4. 交互进程优化

Kernel 2.6 为了增加系统在高负载情况下的交互感受，做了以下三点优化。

(1) interactive_credit ——奖励 sleep_avg

interactive_credit 是设置在 task_struct 里面用来标记进程的“交互程度”的，它在进程创建时候被置为 0，以后随着不同的情况而增加，减少。

- ✧ 增加

  interactive_credit 有两处增 1 的地方，都在函数 recalc_task_prio()里面。

  a. 进程所拥有的内存区域不为空(p->mm!=NULL)，即进程不是内核进程，如果不是从 TASK_UNINTERRUPTIBLE 状态中被唤醒的（p->activated!=-1），且等待的时间（包括在休眠中等待时间和在就绪队列中等待时间）超过了一定限度（sleep_time>INTERACTIVE_SLEEP(p)）；此时将 interactive_credit 增 1；

  b. 进程的等待时间大于 NS_MAX_SLEEP_AVG 了，这种进程很可能是交互进程，所以 interactive_credit 增 1。

- ✧ 减少

  interactive_credit 只有一处地方减 1，在函数 schedule()里面。当进程将要被切换出 CPU 的时候，要计算进程的运行时间 run_time，并将进程的 sleep_avg 进行调整，如果调整后的 sleep_avg 小于 0（说明进程的运行时间大于等待时间），而且该进程的 interactive_credit 在 HIGH_CREDIT(p)和 LOW_CREDIT(p)之间（说明该进程非交互进程），则将 interactive_credit 减 1 作为对进程的惩罚。

从上面的分析可以看出，无论 interactive_credit 如何增减，它都在-(CREDIT_LIMIT+1) ~ (CREDIT_LIMIT+1)范围内；而且当 interactive_credit 增大到 CREDIT_LIMIT+1，即调度器认定该进程为交互进程以后，interactive_credit 就不再变化。

调度器采用宏 HIGH_CREDIT()来判断一个进程是否是交互进程，如果是，则该进程将得到以下奖励：

- ✧ 当进程被剥夺 CPU 使用权时，如果发现该进程是交互进程，则将该进程的运行时间减小，run_time /= (CURRENT_BONUS(prev) ? : 1)。即 sleep_avg 减去的运行时间比实际的运行时间要小，从而增加进程的 sleep_avg。
- ✧ 交互式进程在就绪队列上等待的时间也将增加到 sleep_avg 里面，p->sleep_avg += sleep_time；从而增加进程的 sleep_avg。

可见，对于交互进程都是奖励 sleep_avg 的，从而达到提高优先级的目的。对于交互式进程，调度器并没有在时间片上进行奖励，而是在优先级上进行奖励，是因为交互式进程通常是运行时间短、睡眠时间长，而且要求响应快，而奖励优先级可以给交互进程更多的运行机会，因此，调度器对于交互进程的奖励办法是非常公平和科学的。

(2) 平均等待时间 sleep_avg ——奖励动态优先级

在“平均等待时间”一节已做详细介绍。对于交互进程来说，因为它睡眠的时间较长，所以 sleep_avg 要大一些。另外，经常处于 TASK_INTERRUPTIBLE 状态，而且是被中断唤醒的进程最有可能是交互进程，而这种进程的衡量因素也是 sleep_avg。

总之，由于交互进程一般 sleep_avg 较大，所以调度器通过奖励动态优先级的方式来使得进程获得更多执行的机会。

(3) TASK_INTERACTIVE() ——奖励再次被插入 active array

这个宏是根据进程的动态优先级和静态优先级来判断该进程的“交互程度”。在进程时间片用完时，使用这个宏作为一个参考因素来决定是否将进程重新插入 active array。它的定义是：

```
(p)->prio <= (p)->static_prio - DELTA(p)
DELTA(p) =    (SCALE(TASK_NICE(p), 40, MAX_BONUS) + INTERACTIVE_DELTA)
```

```
SCALE(v1,v1_max,v2_max) = (v1) * (v2_max) / (v1_max)
```

程序　宏 TASK_INTERACTIVE()的定义

可以看出这个宏是将进程的动态优先级和进程的静态优先级做比较，以判断 nice 值为 n（静态优先级）时，进程 p 需要多大的动态优先级才能具有“足够的交互性”。从宏的定义可以看出当进程的 nice 值大于 12 时，进程是不可能被认为是具有足够的交互性（因为 nice>12 时，DELTA(p)>5，而由于 sleep_avg 给进程带来的动态优先级上的奖励最大只有 5，所以 TASK_INTERACTIVE(p)永假）；当进程的 nice 值为-20 时，进程的 sleep_avg 必须非常小才可能使得 TASK_INTERACTIVE(p)值为假。

从以上分析可以看出，这三种奖励办法一个比一个奖励力度大，奖励条件也一个比一个苛刻。而且调度器将用户的意愿放在了第一位（因为 nice 值是可以通过系统调用改变的），由于用户的意愿而给予的奖励（再次被插入 active array）最大，而调度器所给予的奖励占的比例并不大。

5. 调度器主函数 schedule()（kernel/sched.c）

schedule()是用来挑选出下一个应该执行的进程，并且完成进程切换的工作，是进程调度的主要执行者，也是操作系统 Kernel 很重要的一个函数，它的性能将直接决定操作系统的性能。

(1) 函数主要流程

✧ 两个重要数据：prev 和 next

prev　　当前进程，也就是即将被切换出 CPU 的进程

next　　下一个进程，也就是即将被切换进 CPU 的进程

✧ 准备工作

a. 做原子操作方面的检查（主要是检查内核抢占和内核锁的深度是否一致）；

b. 关闭内核抢占（通过函数 preempt_disable()，详见“内核可抢占”一节），因为此时将要对内核一系列重要数据进行操作，所以必须将内核抢占关闭；

c. 将当前进程 current 赋值给 prev，获取当前 CPU 的运行队列 rq，释放 prev 的内核锁（因为即将对 prev 做一系列操作），计算 prev 的运行时间（如果是交互进程则给予 run_time 上的奖励，详见“interactive_credit”一节），给 rq 上自旋锁（防止其他进程访问 rq）；

d. 进行内核的数据统计（如上下文切换次数等），如果 prev 处于可中断状态，而且有信号等待处理，则将 prev 状态置为 TASK_RUNNING，否则将 prev 从 rq 中删除。（这一部分的代码主要是因为在进程在转入睡眠状态时，需要主动调用 schedule()函数）；

e. 如果 rq 中就绪进程个数为 0，而且系统是 SMP，则进行负载均衡的操作（详见“负载均衡”一节），否则将 next 置为 idle 进程，赋值 rq->expired_timestamp = 0（具体含义参见“expired_timestamp”的介绍一节），然后直接进行进程切换。

✧ 寻找最高优先级进程

a. 如果 rq 的 active array 中进程个数为 0，则将 active array 和 expired array 进行切换。具体的过程由以下代码完成：

```
array = rq->active;
rq->active = rq->expired;
rq->expired = array;
rq->expired_timestamp = 0;
```

```
rq->best_expired_prio = MAX_PRIO;
```

程序　切换 active array 和 expired array

b. 用函数 sched_find_first_bit()找到优先级最高的进程队列的偏移量 idx，那么 queue[idx]->next 即为所找的 next，可以通过以下三行代码快速完成：

```
idx = sched_find_first_bit(array->bitmap);
queue = array->queue + idx;
next = list_entry(queue->next, task_t, run_list);
```

程序　寻找最高优先级进程

具体的过程可以形象的表示为：

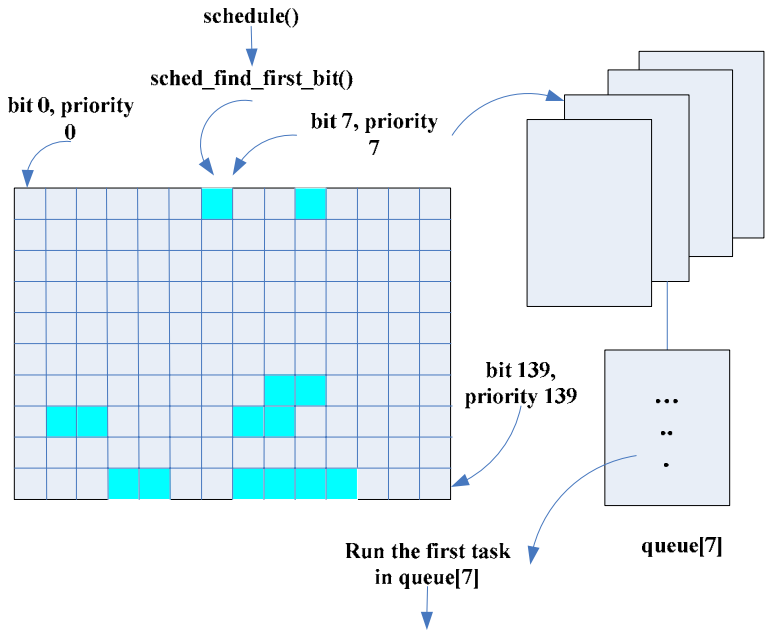

图 3 寻找最高优先级进程

c. 如果 next 是从 TASK_INTERRUPTIBLE 状态中被唤醒的（actived>0），则将进程从就绪队列中删除，将进程在就绪队列上的等待时间也加在等待时间里面重新计算进程的 prio（详见“平均等待时间”一节），再根据新的优先级将进程插入相应就绪队列。

✧ 进程切换

a. 如果 prev!=next，则进行进程切换；

b. 进行进程切换前的准备：将当前时间赋给 next->timestamp，并且将 rq->curr=next；可见此时的 rq->curr 与 current 不再相同；

c. 进程切换，包括内存、堆栈切换等。具体过程和 Kernel 2.4 大致相同，在这里不再赘述；

✧ 完成进程切换后

完成进程切换过后，还需要进行释放 prev 的 mm，给 rq 解锁，重新给 current 获得内核锁（注意在此时 current=next=rq->curr），使能内核抢占；最后检查 TIF_NEED_RESCHED 位，如果已被置位，则重新开始进行调度，重复上述过程；否则调度结束。

(2) 函数执行时机

schedule()函数何时被调用，如何被调用也是一个非常重要的问题。

在 Kernel 2.4 里面，schedule()函数可以通过两种方式调用：

✧ 一种是主动调度，直接调用函数 schedule()，如进程退出，或者进入睡眠状态等。

- ✧ 一种是强制性调度，置位当前进程 task_struct 里面的 need_resched。当是从内核态返回用户态的时候将检查这个位，如果发现已经被置位，会调用 schedule()；有以下三种情况可能会置位 need_resched：
  a. 时钟中断服务程序中，发现进程已经用完自己的时间片，需要被切出 CPU；
  b. 当唤醒一个睡眠进程时，发现该进程比当前占有 CPU 的进程更有运行资格；
  c. 一个进程通过系统调用改变调度政策、nice 值等。
  和主动调度相比，强制性调度有一定的调度延时。

Kernel2.6 的调度时机包含了 Kernel 2.4 的调度时机（不同的就是 need_resched 变成了一个 bit）同时加入了一个重要的特性——内核可抢占，具体的分析见“内核可抢占”一节。

6. 进程调度的生与死

这一部分分析了系统调度器开始工作的时机，以及一个进程从创建到灭亡过程中和进程调度相关的信息和函数。

(1) 系统启动时进程调度的初始化 —— sched_init()

系统进程调度的初始化由 sched_init()函数完成，它被 init/main.c 中函数 start_kernel()调用，该函数主要完成以下工作：

- ✧ 对于所有的 CPU，完成 runqueue 的初始化工作；
- ✧ 对于 SMP，要获取第一个进程的 CPU 号；
- ✧ 调用 wake_up_forked_process()（参见下面“wake_up_forked_process”一节）来唤醒当前进程；
- ✧ 初始化 timer

(2) 创建新进程时的调度信息改变 —— sched_fork(task_t *p)

当当前进程 fork 出一个新进程的时候，需要改变新进程的调度信息，该函数主要的调用关系是：kenel/fork.c – do_fork()->copy_process->sched_fork()，函数主要完成：

- ✧ 将进程状态置为 TASK_RUNNIG，但并未将它加入 runqueue，主要是为了保证没有其他人运行该程序，并且信号或者其他外部事件都不能将它唤醒；
- ✧ 初始化进程的 runlist、array、自旋锁（开子进程的自旋锁，直到 fork 结束，返回用户态时调用函数 sched_tail 来解锁），preempt_count 赋 1；
- ✧ 将子进程的 first_timeslice 置 1，标志这是子进程第一次分配到时间片；
- ✧ 将父进程时间片的一半赋给子进程，同时父进程的时间片减半（这样是为了防止一个进程通过不停的 fork 出子进程来占有 CPU）；如果父进程的时间片此时变为 0，则将其时间片置为 1，相当于此时父进程即将用完其时间片，调用 scheduler_tick()来开始新的调度（具体见“scheduler_tick”一节）。

(3) 初始化新进程的统计信息 —— wake_up_forked_process(task_t * p)

该函数是每一个刚被 fork 出来的进程必须执行的函数，被 kernel/fork.c 中的 do_fork()函数调用，函数主要完成一些 fork 出的新进程统计信息的初始化，主要包括：

- ✧ 父进程和子进程 sleep_avg 的变化（请参照“sleep_avg 进程创建”一节）；
- ✧ 子进程的 interactive_credit 置为 0，重新计算子进程的 prio，设置子进程的 cpu 号；
- ✧ 如果当前进程不在任何 active array 中（如 idle 进程），则调用__activate_task(p, rq)将子进程加入到 active array 里面；否则将父进程的动态优先级赋给子进程，并且将子进程添加到父进程的运行队列中去。

(4) 创建进程完毕 —— schedule_tail()

这个函数是在 fork()系统调用即将完成，返回用户态之前，经过 entry.S 时调用的。函数主要完成一些 fork 完毕需做的清理工作，如释放上文所说的自旋锁等。

(5) 进程运行过程中 —— scheduler_tick()

update_process_time()（被时钟中断服务程序调用）调用该函数来更新当前进程的时间片，并且根据减小后的结果进行相应的处理。函数主要完成：

- ✧ 完成当前进程使用的系统时间、用户时间的统计信息；
- ✧ 如果当前进程是实时进程，调度策略是 SCHED_RR（调度策略是 SCHED_FIFO 的进程不需要重新分配时间片），且已经用完时间片，则重新计算时间片，将（表明该进程退出时不会把时间片交还给父进程），置位 TIF_NEED_RESCHED，将进程放到进程队列的末尾；
- ✧ 如果不是实时进程，且用完时间片：

  a. 置位 TIF_NEED_RESCHED，重新计算进程的动态优先级和时间片，将 first_timeslice 置 0，记录 rq-> expired_timestamp 的值（意义参见“ expired_timestamp ”一节）；

  b. 根据 TASK_INTERACTIVE()（宏的意义参见“ TASK_INTERACTIVE ”一节）判断是否交互进程，用宏 EXPIRED_STARVING(rq)判断 expired array 是否已经饥饿，将该宏展开后为：

```
(STARVATION_LIMIT && ((rq)->expired_timestamp&&(jiffies-
(rq)->expired_timestamp >= STARVATION_LIMIT * ((rq)->nr_running) + 1)))
|| ((rq)->curr->static_prio > (rq)->best_expired_prio)
```

程序 宏 EXPIRED_STARVING()

  可见如果 EXPIRED_STARVING()的是否为真与三个因素有关：

  - ✧ STARVATION_LIMIT 为真；
  - ✧ (rq)->expired_timestamp 为真；
  - ✧ 若(rq)->expired_timestamp >= STARVATION_LIMIT * ((rq)->nr_running) + 1)（说明 expired array 上的进程已经等了足够长的时间）为真，或者((rq)->curr->static_prio > (rq)->best_expired_prio)（说明当前进程的静态优先级比 expired array 中最高的优先级低）为真。

  c. 如果进程被认为是交互进程，而且 EXPIRED_STARVING()为假，则将当前进程重新插入到 active array 里面（参见“ TASK_INTERACTIVE ”一节）；否则，将进程插入到 expired array。

- ✧ 如果进程尚未用完时间片，该进程是交互式进程，且剩余的时间片是该进程时间片粒度的整数倍（至少 1 倍），则强行剥夺该进程 CPU 使用权，且放到 active array 里面运行队列的末尾（实际上是在交互式进程内部实行 RR 策略了）。

  时间片粒度的和两个因素有关：

  a. sleep_avg sleep_avg 越大，粒度越大，因为越大说明该进程是交互进程的可能性越大，交互式进程的特点就是时间片小，频率高；反之，如果是一个 CPU-bound 进程就应该少分片或者不分片（尽量避免 cache 失配），应该有高的粒度；

  b. CPU 个数 CPU 个数越多，运行粒度就越大。

(6) 进程状态的相互转换（sleep 和 wake up）

进程在 interruptible 和 running 状态之间切换的关系可以简单的用下图表示：

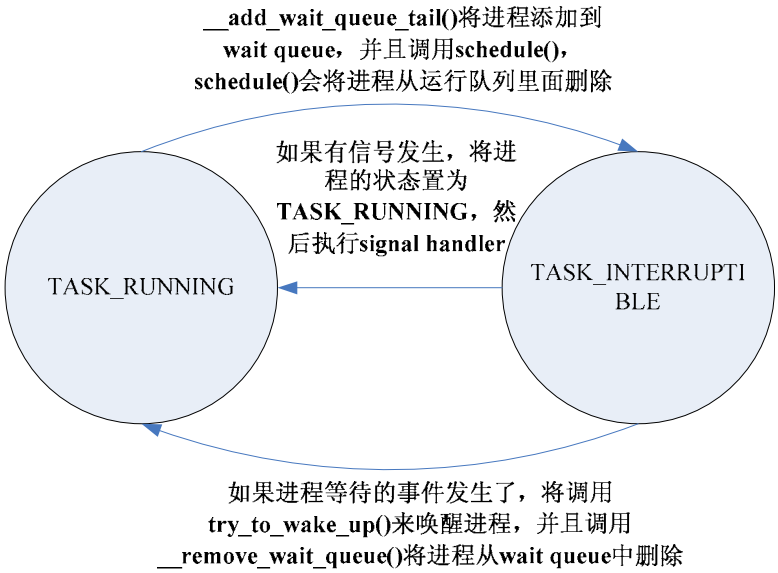

图 4　进程在 TASK_INTERRUPTBLE 和 TASK_RUNNING 之间切换

这里简单的介绍函数 wait_for_completion()和 try_to_wake_up()。

✧ wait_for_completion()

该函数是标准的将用户由就绪状态转为睡眠状态的函数，主要经过以下几个步骤：

a. 通过 DECLARE_WAITQUEUE()创建一个等待队列入口；

b. 通过函数__add_wait_queue_tail()将进程加入到等待队列；

c. 将进程状态置为 TASK_UNINTERRUPTIBLE；

d. 调用 schedule()函数进行调度；

e. 利用循环检查进程等待条件是否满足，如果满足通过函数__remove_wait_queue()将进程从等待队列中删除；否则，继续通过 schedule()进行调度。

✧ try_to_wake_up()

函数调用 activate_task()将进程加到就绪队列中，如果新唤醒的进程的优先级比 rq->curr 高（具体原因请参见“curr”一节），则置位 TIF_NEED_RESCHED，最后将进程的状态置为 TASK_RUNNING。

(7) 进程结束，退出调度 —— sched_exit(task_t *p)

被 release_task()调用，用于处理进程销毁前调度信息的清理，包括：

✧ 根据 p->first_timeslice 来判断是否将时间片交还给父进程；如果 first_timeslice 的值为 1，则说明 p 尚未用完 fork 时从父进程分来的时间片，此时应该将时间片交还父进程；否则，说明子进程已经重新分配过时间片，无须交还；

✧ 如果子进程（p）的执行时间过长（p->sleep_avg < p->parent->sleep_avg），则给予父进程一定的惩罚（稍稍减小父进程的 sleep_avg）。

7. 内核可抢占

Kernel 2.6 的一大亮点就是内核可抢占，是 Kernel 2.6 进程调度优于 2.4 的一个重要表现。

(1) 何时可以抢占内核

在前面我们已经讲了内核何时可以抢占：当内核进程没有访问内核的关键数据，也就是内核没有被加锁，此时内核代码是可重入的，可以抢占内核。

对内核抢占加锁是通过 preempt_disable()来实现的，这个宏只是简单的将 preempt_count 增 1 就实现了内核的加锁，表明此时已经进入内核的关键数据区域，内核不可被抢占。

(2) 如何抢占内核

✧ 中断返回内核时

在前面介绍“ preempt_count ”的时候已经提到，内核能否抢占是通过操作 preempt_count 来实现的。注意 arch/i386/kernel/entry.S 里面以下程序：

```
ENTRY(resume_kernel)
        cmpl $0,TI_PRE_COUNT(%ebp) # non-zero preempt_count ?
        jnz restore_all
need_resched:
        movl TI_FLAGS(%ebp), %ecx      # need_resched set ?
        testb $_TIF_NEED_RESCHED, %cl
        jz restore_all
        testl $IF_MASK,EFLAGS(%esp)         # interrupts off (exception path) ?
        jz restore_all
        movl $PREEMPT_ACTIVE,TI_PRE_COUNT(%ebp)
        sti
        call schedule
        movl $0,TI_PRE_COUNT(%ebp)
        cli
        jmp need_resched
```

程序　内核抢占入口代码

程序中可以看出，在中断或者异常返回内核空间以后，首先检查 preempt_count 是否为 0，如果不为 0，说明已经内核已经禁止被抢占；如果为 0，则检查 TIF_NEED_RESCHED 位，如果已经被置位则检查此次是否是通过中断（通过检查堆栈中 EFLAGS 的 IF 位来检查发生此次“ 中断 ”前 IF 是否被置位，如果被置位说明是中断；否则说明是由异常返回内核）返回内核空间的，如果是，则调用 schedule()函数进行调度。可见，抢占内核是发生在由中断返回内核空间的时候。

✧ 解锁时

解锁通过宏 preempt_enable()来完成，此函数完成以下功能：

a. 将当前进程的 preempt_count 减 1

b. 检查 TIF_NEED_RESCHED 位，如果是 0，则返回；否则调用函数 preempt_schedule()，此函数将 preempt_count 置为 PREEMPT_ACTIVE（表明正在执行内核抢占），然后直接调用 schedule()进行调度。

✧ 内核代码中直接调用函数 schedule()

这种情况下是没有任何保护措施的，也就是说调用的代码必须清楚此时进行内核抢占是否安全。

8. 负载均衡

Kernel 2.6 的负载均衡分为两种，一种是“ pull ”，一种是“ push ”。

(1) pull

当一个 CPU 负载轻，而另外一个 CPU 负载过重的时候，调度器会从负载重的 CPU 把进程 pull 过来，这个过程主要通过函数 load_balance()完成。

load_balance()有两种工作方式，一种是当前 CPU 完全空闲，idle=1，另外一种是上面仍有进程在运行，idle=0；当 idle=1 时，很多操作将变得很简单。

当 idle=0 时，不管当前 CPU 有多忙碌，定时器都将定期启动函数 rebalance_tick()，而该函数将每隔 BUSY_REBALANCE_TICK 时间就调用函数 load_balance()来进行负载均衡。load_balance()的函数流程如下：

- ✧ 找到最忙的 CPU：取当前 CPU 负荷为当前负荷和历史负荷里面的最大者，取其他 CPU 负荷为当前负荷和历史符合里面最小者；最忙的 CPU 的负荷必须比当前 CPU 的负荷高 25%，否则不进行迁移；对源、目的两个就绪队列加锁之后，再次检查源就绪队列负载是否减小，如果是则退出负载均衡；
- ✧ 找到最忙 CPU 后，按照从 expired 队列到 active 队列，从高优先级到低优先级进程进行迁移，不过以下几种进程不进行迁移：
  a. 当前正在执行的进程；
  b. 通过 cpus_allowed 明确表示不能迁移该 CPU 的进程；
  c. 被原来 CPU 切换下来的时间小于 cache_decay_ticks（说明 cache 仍然活跃）。
- ✧ 进行任务迁移，主要进行以下操作
  a. 将进程从原来 CPU 的相应就绪队列中删除；
  b. 将进程的 CPU 号设置为当前进程，并将任务添加到当前 CPU 的 active array 中；
  c. 赋值 timestamp；
  d. 如果新迁移过来的任务比当前 CPU 正在运行的程序优先级更高，则置位 TIF_NEED_RESCHED。
- ✧ 如果此时负载仍然没有平衡（通过 imbalance 的值来反映），则重复上述过程，直到平衡为止。

当 idle=1 时，有两个时机调用 load_balance()函数：

- ✧ 定时器每隔 IDLE_REBALANCE_TICK 时间启动 load_balance()函数；
- ✧ 在 schedule()函数里，如果发现该 CPU 上 rq 为空，则主动调用 load_balance()函数进行负载均衡。

idle 时候调用 load_balance()的流程和 idle=0 是一样，只不过将 idle 作为参数传进去，idle=1 可以简化很多判断，比如每次只迁移一个进程等。

(2) push

主要通过 migration_thread()这个核心进程来将本 CPU 上的 rq->migration_queue“ push ”到别的 CPU 上，在 Kernel 2.6 里，是选择第一个允许运行的 CPU。该进程是调度策略为 SCHED_FIFO 的实时核心进程，一般时候处于睡眠状态。

migration_queue 是通过函数 set_cpus_allowed()来改变运行的 CPU，该函数构造一个类型为 migration_req_t 的迁移队列，将其植入进程所在 CPU 的 migration_queue。然后唤醒 migration_thread 这个核心进程，由它来完成进程的迁移。

migration_thread()通过函数 move_task_away()来完成实际的迁移工作，该函数主要完成以下工作：

- ✧ 设置进程 CPU 号；
- ✧ 将进程从原来的就绪队列里面删除，并用函数 activate_task()来将进程加入目标 CPU 的就绪队列中；
- ✧ 如果迁移的进程比目标 CPU 的 rq->curr 的优先级要高，就调用函数 resched_task()来将目标 CPU 上的当前进程的 TIF_NEED_RESCHED 置为 1。

第三章 对 Kernel 2.6 的一点改进

略

参考资料:

[1] Linus Tovalds, Linux 内核源码 v2.6.4, from www.kernel.org
[2] Robert Love, Linux Kernel Development
[3] 杨沙洲, Linux 2.6 调度系统分析