进程调度有很多方法，这里只讨论Linux下的进程调度，先说下，这个是高端面试必考题，既然我发文了，大家最好看看，而且目前看到的写得最好的文章，推荐给大家。

====

Linux是一个支持多任务的操作系统，而多个任务之间的切换是通过 调度器 来完成，调度器 使用不同的调度算法会有不同的效果。

Linux2.4版本使用的调度算法的时间复杂度为O(n)，其主要原理是通过轮询所有可运行任务列表，然后挑选一个最合适的任务运行，所以其时间复杂度与可运行任务队列的长度成正比。

而Linux2.6开始替换成名为 O(1)调度算法，顾名思义，其时间复杂度为O(1)。虽然在后面的版本开始使用 CFS调度算法（完全公平调度算法），但了解 O(1)调度算法 对学习Linux调度器还是有很大帮助的，所以本文主要介绍 O(1)调度算法 的原理与实现。

由于在 Linux 内核中，任务和进程是相同的概念，所以在本文混用了任务和进程这两个名词。

**O(1)调度算法原理**

**prio\_array 结构**

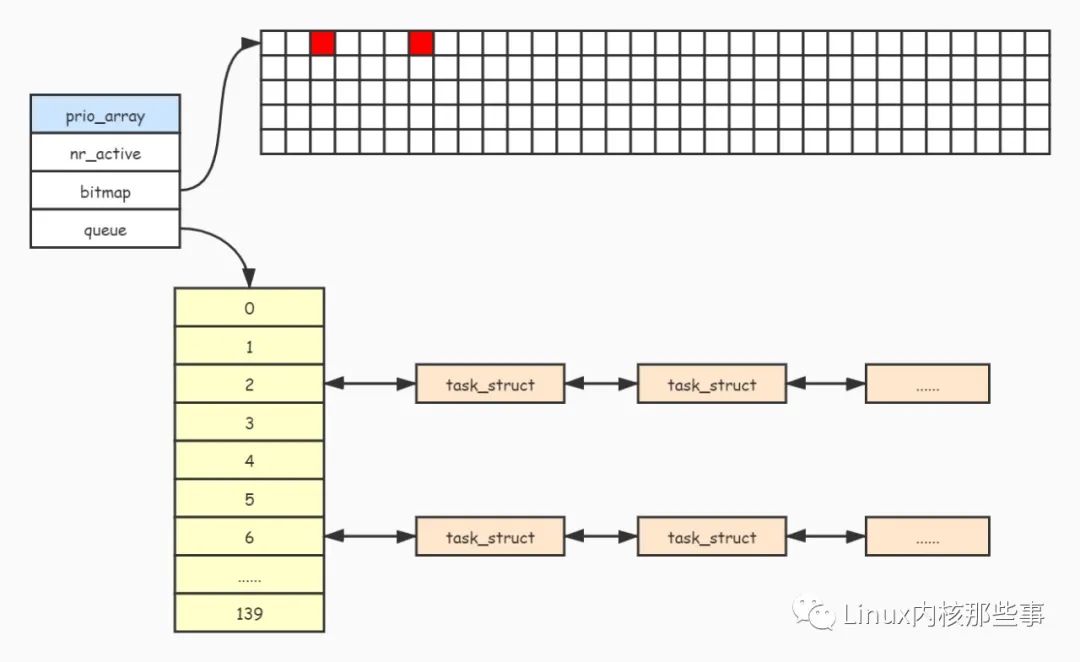
O(1)调度算法 通过优先级来对任务进行分组，可分为140个优先级（0 ~ 139，数值越小优先级越高），每个优先级的任务由一个队列来维护。prio\_array 结构就是用来维护这些任务队列，如下代码：

#define MAX\_USER\_RT\_PRIO 100  
#define MAX\_RT\_PRIO MAX\_USER\_RT\_PRIO  
#define MAX\_PRIO (MAX\_RT\_PRIO + 40)  
  
#define BITMAP\_SIZE ((((MAX\_PRIO+1+7)/8)+sizeof(long)-1)/sizeof(long))  
  
struct prio\_array {  
 int nr\_active;  
 unsigned long bitmap[BITMAP\_SIZE];  
 struct list\_head queue[MAX\_PRIO];  
};

下面介绍 prio\_array 结构各个字段的作用：

1. nr\_active: 所有优先级队列中的总任务数。
2. bitmap: 位图，每个位对应一个优先级的任务队列，用于记录哪个任务队列不为空，能通过 bitmap 够快速找到不为空的任务队列。
3. queue: 优先级队列数组，每个元素维护一个优先级队列，比如索引为0的元素维护着优先级为0的任务队列。

下图更直观地展示了 prio\_array 结构各个字段的关系：



如上图所述，bitmap 的第2位和第6位为1（红色代表为1，白色代表为0），表示优先级为2和6的任务队列不为空，也就是说 queue 数组的第2个元素和第6个元素的队列不为空。

**runqueue 结构**

另外，为了减少多核CPU之间的竞争，所以每个CPU都需要维护一份本地的优先队列。因为如果使用全局的优先队列，那么多核CPU就需要对全局优先队列进行上锁，从而导致性能下降。

每个CPU都需要维护一个 runqueue 结构，runqueue 结构主要维护任务调度相关的信息，比如优先队列、调度次数、CPU负载信息等。其定义如下：

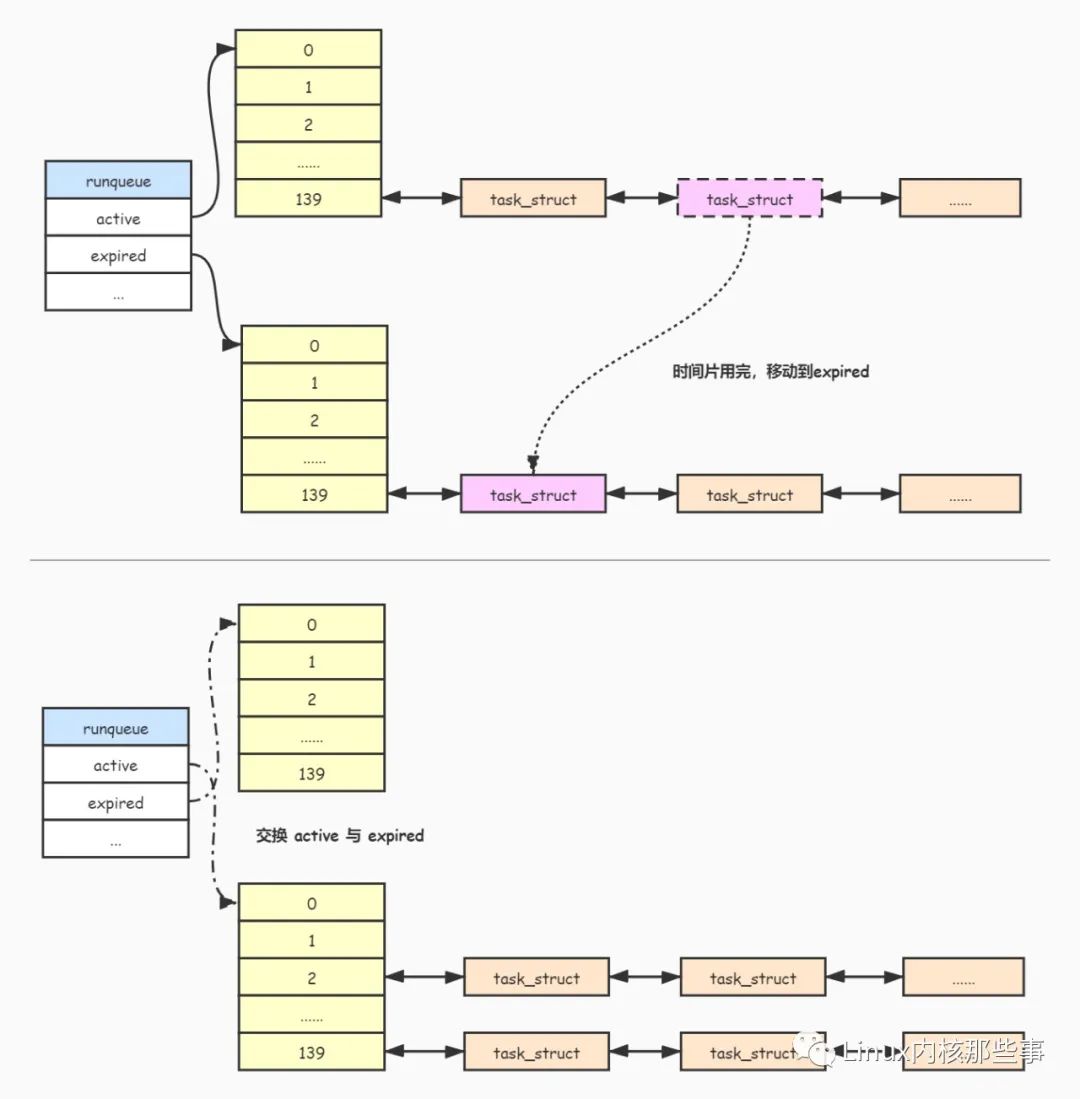
struct runqueue {  
 spinlock\_t lock;  
 unsigned long nr\_running,  
 nr\_switches,  
 expired\_timestamp,  
 nr\_uninterruptible;  
 task\_t \*curr, \*idle;  
 struct mm\_struct \*prev\_mm;  
 prio\_array\_t \*active, \*expired, arrays[2];  
 int prev\_cpu\_load[NR\_CPUS];  
 task\_t \*migration\_thread;  
 struct list\_head migration\_queue;  
 atomic\_t nr\_iowait;  
};

runqueue 结构有两个重要的字段：active 和 expired，这两个字段在 O(1)调度算法 中起着至关重要的作用。我们先来了解一下 O(1)调度算法 的大概原理。

我们注意到 active 和 expired 字段的类型为 prio\_array，指向任务优先队列。active 代表可以调度的任务队列，而 expired 字段代表时间片已经用完的任务队列。active 和 expired 会进行以下两个过程：

1. 当 active 中的任务时间片用完，那么就会被移动到 expired 中。
2. 当 active 中已经没有任务可以运行，就把 expired 与 active 交换，从而 expired 中的任务可以重新被调度。

**如下图所示：**



O(1)调度算法 把140个优先级的前100个（0 ~ 99）作为 实时进程优先级，而后40个（100 ~ 139）作为 普通进程优先级。实时进程被放置到实时进程优先级的队列中，而普通进程放置到普通进程优先级的队列中。

**实时进程调度**

实时进程分为 FIFO（先进先出） 和 RR（时间轮询） 两种，其调度算法比较简单，如下：

1. 先进先出的实时进程调度：如果调度器在执行某个先进先出的实时进程，那么调度器会一直运行这个进程，直至其主动放弃运行权（退出进程或者sleep等）。
2. 时间轮询的实时进程调度：如果调度器在执行某个时间轮询的实时进程，那么调度器会判断当前进程的时间片是否用完，如果用完的话，那么重新分配时间片给它，并且重新放置回 active 队列中，然后调度到其他同优先级或者优先级更高的实时进程进行运行。

**普通进程调度**

每个进程都要一个动态优先级和静态优先级，静态优先级不会变化（进程创建时被设置），而动态优先级会随着进程的睡眠时间而发生变化。动态优先级可以通过以下公式进行计算：

动态优先级 = max(100, min(静态优先级 – bonus + 5), 139))

上面公式的 bonus（奖励或惩罚） 是通过进程的睡眠时间计算出来，进程的睡眠时间越大，bonus 的值就越大，那么动态优先级就越高（前面说过优先级的值越小，优先级越高）。

另外要说明一下，实时进程的动态优先级与静态优先级相同。

当一个普通进程被添加到运行队列时，会先计算其动态优先级，然后按照动态优先级的值来添加到对应优先级的队列中。而调度器调度进程时，会先选择优先级最高的任务队列中的进程进行调度运行。

**运行时间片计算**

当进程的时间用完后，就需要重新进行计算。进程的运行时间片与静态优先级有关，可以通过以下公式进行计算：

静态优先级 < 120，运行时间片 = max((140-静态优先级)\*20, MIN\_TIMESLICE)  
静态优先级 >= 120，运行时间片 = max((140-静态优先级)\*5, MIN\_TIMESLICE)

**O(1)调度算法实现**

接下来我们分析一下 O(1)调度算法 在内核中的实现。

**时钟中断**

时钟中断是由硬件触发的，可以通过编程来设置其频率，Linux内核一般设置为每秒产生100 ~ 1000次。时钟中断会触发调用 scheduler\_tick() 内核函数，其主要工作是：减少进程的可运行时间片，如果时间片用完，那么把进程从 active 队列移动到 expired 队列中。代码如下：

void scheduler\_tick(int user\_ticks, int sys\_ticks)  
{  
 runqueue\_t \*rq = this\_rq();  
 task\_t \*p = current;  
  
 ...  
  
 // 处理普通进程  
 if (!--p->time\_slice) { // 减少时间片, 如果时间片用完  
 dequeue\_task(p, rq->active); // 把进程从运行队列中删除  
 set\_tsk\_need\_resched(p); // 设置要重新调度标志  
 p->prio = effective\_prio(p); // 重新计算动态优先级  
 p->time\_slice = task\_timeslice(p); // 重新计算时间片  
 p->first\_time\_slice = 0;  
  
 if (!rq->expired\_timestamp)  
 rq->expired\_timestamp = jiffies;  
  
 // 如果不是交互进程或者没有出来饥饿状态  
 if (!TASK\_INTERACTIVE(p) || EXPIRED\_STARVING(rq)) {  
 enqueue\_task(p, rq->expired); // 移动到expired队列  
 } else  
 enqueue\_task(p, rq->active); // 重新放置到active队列  
 }  
 ...  
}

上面代码主要完成以下几个工作：

1. 减少进程的时间片，并且判断时间片是否已经使用完。
2. 如果时间片使用完，那么把进程从 active 队列中删除。
3. 调用 set\_tsk\_need\_resched() 函数设 TIF\_NEED\_RESCHED 标志，表示当前进程需要重新调度。
4. 调用 effective\_prio() 函数重新计算进程的动态优先级。
5. 调用 task\_timeslice() 函数重新计算进程的可运行时间片。
6. 如果当前进程是交互进程或者出来饥饿状态，那么重新加入到 active 队列。
7. 否则把今天移动到 expired 队列。

**任务调度**

如果进程设置了 TIF\_NEED\_RESCHED 标志，那么当从时钟中断返回到用户空间时，会调用 schedule() 函数进行任务调度。schedule() 函数代码如下：

void schedule(void)  
{  
 ...  
 prev = current; // 当前需要被调度的进程  
 rq = this\_rq(); // 获取当前CPU的runqueue  
  
 array = rq->active; // active队列  
  
 // 如果active队列中没有进程, 那么替换成expired队列  
 if (unlikely(!array->nr\_active)) {  
 rq->active = rq->expired;  
 rq->expired = array;  
 array = rq->active;  
 rq->expired\_timestamp = 0;  
 }  
  
 idx = sched\_find\_first\_bit(array->bitmap); // 找到最高优先级的任务队列  
 queue = array->queue + idx;  
 next = list\_entry(queue->next, task\_t, run\_list); // 获取到下一个将要运行的进程  
  
 ...  
 prev->sleep\_avg -= run\_time; // 减少当前进程的睡眠时间  
 ...  
  
 if (likely(prev != next)) {  
 ...  
 prev = context\_switch(rq, prev, next); // 切换到next进程进行运行  
 ...  
 }  
 ...  
}

上面代码主要完成以下几个步骤：

1. 如果当前 runqueue 的 active 队列为空，那么把 active 队列与 expired 队列进行交换。
2. 调用 sched\_find\_first\_bit() 函数在 bitmap 中找到优先级最高并且不为空的任务队列索引。
3. 减少当前进程的睡眠时间。
4. 调用 context\_switch() 函数切换到next进程进行运行。