Linux 进程调度原理

进程调度依据

调度程序运行时，要在所有可运行状态的进程中选择最值得运行的进程投入运行。选择进程的依据是什么呢？在每个进程的 task_struct 结构中有以下四项：policy、priority、counter、rt_priority。这四项是选择进程的依据。其中，policy 是进程的调度策略，用来区分实时进程和普通进程，实时进程优先于普通进程运行；priority 是进程(包括实时和普通)的静态优先级；counter 是进程剩余的时间片，它的起始值就是 priority 的值；由于 counter 在后面计算一个处于可运行状态的进程值得运行的程度 goodness 时起重要作用，因此，counter 也可以看作是进程的动态优先级。rt_priority 是实时进程特有的，用于实时进程间的选择。

Linux 用函数 goodness()来衡量一个处于可运行状态的进程值得运行的程度。该函数综合了以上提到的四项，还结合了一些其他的因素，给每个处于可运行状态的进程赋予一个权值(weight)，调度程序以这个权值作为选择进程的唯一依据。关于 goodness()的情况在后面将会详细分析。

进程调度策略

调度程序运行时,要在所有处于可运行状态的进程之中选择最值得运行的进程投入运行。选择进程的依据是什么呢?在每个进程的 task_struct 结构中有这么四项：

policy, priority , counter, rt_priority

这四项就是调度程序选择进程的依据.其中,policy 是进程的调度策略,用来区分两种进程-实时和普通；priority 是进程(实时和普通)的优先级；counter 是进程剩余的时间片,它的大小完全由 priority 决定;rt_priority 是实时优先级,这是实时进程所特有的，用于实时进程间的选择。

首先，Linux 根据 policy 从整体上区分实时进程和普通进程，因为实时进程和普通进程度调度是不同的，它们两者之间，实时进程应该先于普通进程而运行，然后，对于同一类型的不同进程，采用不同的标准来选择进程：

对于普通进程，Linux 采用动态优先调度，选择进程的依据就是进程 counter 的大小。进程创建时，优先级 priority 被赋一个初值，一般为 0～70 之间的数字，这个数字同时也是计数器 counter 的初值，就是说进程创建时两者是相等的。字面上看，priority 是"优先级"、counter 是"计数器"的意思，然而实际上，它们表达的是同一个意思-进程的"时间片"。Priority 代表分配给该进程的时间片，counter 表示该进程剩余的时间片。在进程运行过程中，counter 不断减少，而 priority 保持不变，以便在 counter 变为 0 的时候（该进程用完了所分配的时间片）对 counter 重新赋值。当一个普通进程的时间片用完以后，并不马上用 priority 对 counter 进行赋值，只有所有处于可运行状态的普通进程的时间片(p->counter==0)都用完了以后，才用 priority 对 counter 重新赋值，这个普通进程才有了再次被调度的机会。这说明，普通进程运行过程中，counter 的减小给了其它进程得以运行的机会，直至 counter 减为 0 时才完全放弃对 CPU 的使用，这就相对于优先级在动态变化，所以称之为动态优先调度。至于时间片这个概念，和其他不同操作系统一样的，Linux 的时间单位也是"时钟滴答"，只是不同操作系统对一个时钟滴答的定义不同而已（Linux 为 10ms）。进程的时间片就是指多少个时钟滴答，比如，若 priority 为 20，则分配给该进程的时间片就为 20 个时钟滴答，也就是 20*10ms=200ms。Linux 中某个进程的调度策略(policy)、优先级(priority)等可以作为参数由用户自己决定，具有相当的灵活性。内核创建新进程时分配给进程的时间片缺省为 200ms(更准确的，应为 210ms)，用户可以通过系统调用改变它。

对于实时进程，Linux 采用了两种调度策略，即 FIFO(先来先服务调度)和 RR（时间片轮转

调度)。因为实时进程具有一定程度的紧迫性，所以衡量一个实时进程是否应该运行，Linux 采用了一个比较固定的标准。实时进程的 counter 只是用来表示该进程的剩余时间片，并不作为衡量它是否值得运行的标准。实时进程的 counter 只是用来表示该进程的剩余时间片，并不作为衡量它是否值得运行的标准，这和普通进程是有区别的。上面已经看到，每个进程有两个优先级，实时优先级就是用来衡量实时进程是否值得运行的。

这一切看来比较麻烦，但实际上 Linux 中的实现相当简单。Linux 用函数 goodness()来衡量一个处于可运行状态的进程值得运行的程度。该函数综合了上面提到的各个方面，给每个处于可运行状态的进程赋予一个权值(weight)，调度程序以这个权值作为选择进程的唯一依据。Linux 根据 policy 的值将进程总体上分为实时进程和普通进程，提供了三种调度算法：一种传统的 Unix 调度程序和两个由 POSIX.1b(原名为 POSIX.4)操作系统标准所规定的"实时"调度程序。但这种实时只是软实时，不满足诸如中断等待时间等硬实时要求，只是保证了当实时进程需要时一定只把 CPU 分配给实时进程。

非实时进程有两种优先级，一种是静态优先级，另一种是动态优先级。实时进程又增加了第三种优先级，实时优先级。优先级是一些简单的整数，为了决定应该允许哪一个进程使用 CPU 的资源，用优先级代表相对权值-优先级越高，它得到 CPU 时间的机会也就越大。

? 静态优先级(priority)-不随时间而改变，只能由用户进行修改。它指明了在被迫和其他进程竞争 CPU 之前，该进程所应该被允许的时间片的最大值（但很可能的，在该时间片耗尽之前，进程就被迫交出了 CPU)。

? 动态优先级(counter)-只要进程拥有 CPU，它就随着时间不断减小；当它小于 0 时，标记进程重新调度。它指明了在这个时间片中所剩余的时间量。

? 实时优先级(rt_priority)-指明这个进程自动把 CPU 交给哪一个其他进程；较高权值的进程总是优先于较低权值的进程。如果一个进程不是实时进程，其优先级就是 0，所以实时进程总是优先于非实时进程的（但实际上，实时进程也会主动放弃 CPU)。

当 policy 分别为以下值时：

1) SCHED_OTHER：这是普通的用户进程，进程的缺省类型，采用动态优先调度策略，选择进程的依据主要是根据进程 goodness 值的大小。这种进程在运行时，可以被高 goodness 值的进程抢先。

2) SCHED_FIFO：这是一种实时进程，遵守 POSIX1.b 标准的 FIFO(先入先出)调度规则。它会一直运行，直到有一个进程因 I/O 阻塞，或者主动释放 CPU，或者是 CPU 被另一个具有更高 rt_priority 的实时进程抢先。在 Linux 实现中，SCHED_FIFO 进程仍然拥有时间片-只有当时间片用完时它们才被迫释放 CPU。因此，如同 POSIX1.b 一样，这样的进程就象没有时间片(不是采用分时)一样运行。Linux 中进程仍然保持对其时间片的记录（不修改 counter）主要是为了实现的方便，同时避免在调度代码的关键路径上出现条件判断语句 if (!(current->policy&SCHED_FIFO)){...}-要知道，其他大量非 FIFO 进程都需要记录时间片，这种多余的检测只会浪费 CPU 资源。(一种优化措施，不该将执行时间占 10%的代码的运行时间减少到 50%；而是将执行时间占 90%的代码的运行时间减少到 95%。0.9+0.1*0.5=0.95>0.1+0.9*0.9=0.91)

3) SCHED_RR：这也是一种实时进程，遵守 POSIX1.b 标准的 RR(循环 round-robin)调度规则。除了时间片有些不同外，这种策略与 SCHED_FIFO 类似。当 SCHED_RR 进程的时间片用完后，就被放到 SCHED_FIFO 和 SCHED_RR 队列的末尾。

只要系统中有一个实时进程在运行，则任何 SCHED_OTHER 进程都不能在任何 CPU 运行。每个实时进程有一个 rt_priority，因此，可以按照 rt_priority 在所有 SCHED_RR 进程之间分配 CPU。其作用与 SCHED_OTHER 进程的 priority 作用一样。只有 root 用户能够用系统调用 sched_setscheduler，来改变当前进程的类型(sys_nice,sys_setpriority)。

此外，内核还定义了 SCHED_YIELD，这并不是一种调度策略，而是截取调度策略的一个附加位。如同前面说明的一样，如果有其他进程需要 CPU，它就提示调度程序释放 CPU。特别要注意的就是这甚至会引起实时进程把 CPU 释放给非实时进程。

主要的进程调度的函数分析

真正执行调度的函数是 schedule(void),它选择一个最合适的进程执行，并且真正进行上下文切换，使得选中的进程得以执行。而 reschedule_idle(struct task_struct *p)的作用是为进程选择一个合适的 CPU 来执行，如果它选中了某个 CPU，则将该 CPU 上当前运行进程的 need_resched 标志置为 1,然后向它发出一个重新调度的处理机间中断，使得选中的 CPU 能够在中断处理返回时执行 schedule 函数，真正调度进程 p 在 CPU 上执行。在 schedule()和 reschedule_idle()中调用了 goodness()函数。goodness()函数用来衡量一个处于可运行状态的进程值得运行的程度。此外，在 schedule()函数中还调用了 schedule_tail()函数;在 reschedule_idle()函数中还调用了 reschedule_idle_slow()。这些函数的实现对理解 SMP 的调度非常重要，下面一一分析这些函数。先给出每个函数的主要流程图，然后给出源代码，并加注释。

goodness()函数分析

goodness()函数计算一个处于可运行状态的进程值得运行的程度。一个任务的 goodness 是以下因素的函数：正在运行的任务、想要运行的任务、当前的 CPU。goodness 返回下面两类值中的一个：1000 以下或者 1000 以上。1000 或者 1000 以上的值只能赋给"实时"进程，从 0 到 999 的值只能赋给普通进程。实际上，在单处理器情况下，普通进程的 goodness 值只使用这个范围底部的一部分，从 0 到 41。在 SMP 情况下，SMP 模式会优先照顾等待同一个处理器的进程。不过，不管是 UP 还是 SMP，实时进程的 goodness 值的范围是从 1001 到 1099。goodness()函数其实是不会返回-1000 的，也不会返回其他负值。由于 idle 进程的 counter 值为负，所以如果使用 idle 进程作为参数调用 goodness，就会返回负值，但这是不会发生的。goodness()是个简单的函数，但是它是 linux 调度程序不可缺少的部分。运行队列中的每个进程每次执行 schedule 时都要调度它，因此它的执行速度必须很快。

```
//在/kernel/sched.c 中
static inline int goodness(struct task_struct * p, int this_cpu, struct mm_struct *this_mm)
{ int weight;

if (p->policy != SCHED_OTHER) {/*如果是实时进程，则*/
weight = 1000 + p->rt_priority;
goto out;
}
/*  将 counter 的值赋给 weight，这就给了进程一个大概的权值，counter 中的值表示进程在一个时间片内，剩下要运行的时间.*/
weight = p->counter;
if (!weight) /* weight==0,表示该进程的时间片已经用完，则直接转到标号 out*/
goto out;
#ifdef __SMP__

/*在 SMP 情况下，如果进程将要运行的 CPU 与进程上次运行的 CPU 是一样的，则最有利，因此，假如进程上次运行的 CPU 与当前 CPU 一致的话，权值加上
```

PROC_CHANGE_PENALTY，这个宏定义为 20。*/
if (p->processor == this_cpu)
weight += PROC_CHANGE_PENALTY;
#endif

if (p->mm == this_mm) /*进程 p 与当前运行进程，是同一个进程的不同线程，或者是共享地址空间的不同进程，优先选择，权值加 1*/
weight += 1;
weight += p->priority; /* 权值加上进程的优先级*/
out:
return weight; /* 返回值作为进程调度的唯一依据，谁的权值大，就调度谁运行*/
}
schedule()函数分析
schedule()函数的作用是，选择一个合适的进程在 CPU 上执行，它仅仅根据'goodness'来工作。对于 SMP 情况，除了计算每个进程的加权平均运行时间外，其他与 SMP 相关的部分主要由 goodness()函数来体现。
Schedule()函数的主要流程图如下：

流程：
①将 prev 和 next 设置为 schedule 最感兴趣的两个进程：其中一个是在调用 schedule 时正在运行的进程(prev)，另外一个应该是接着就給予 CPU 的进程（next)。注意：prev 和 next 可能是相同的-schedule 可以重新调度已经获得 cpu 的进程.
②中断处理程序运行"下半部分".
③内核实时系统部分的实现，循环调度程序（SCHED_RR）通过移动"耗尽的"RR 进程-已经用完其时间片的进程-到队列末尾,这样具有相同优先级的其他 RR 进程就可以获得 CPU 了。同时，这补充了耗尽进程的时间片。
④由于代码的其他部分已经决定了进程必须被移进或移出 TASK_RUNNING 状态，所以会经常使用 schedule,例如,如果进程正在等待的硬件条件已经发生,所以如果必要,这个 switch 会改变进程的状态。如果进程已经处于 TASK_RUNNING 状态，它就无需处理了。如果它是可以中断的（等待信号），并且信号已经到达了进程，就返回 TASK_RUNNING 状态。在所以其他情况下（例如，进程已经处于 TASK_UNINTERRUPTIBLE 状态了），应该从运行队列中将进程移走。
⑤将 p 初始化为运行队列的第一个任务；p 会遍历队列中的所有任务。
⑥c 记录了运行队列中所有进程最好的"goodness"-具有最好"goodness"的进程是最易获得 CPU 的进程。goodness 的值越高越好。
⑦遍历执行任务链表，跟踪具有最好 goodness 的进程。
⑧这个循环中只考虑了唯一一个可以调度的进程。在 SMP 模式下，只有任务不在 cpu 上运行时，即 can_schedule 宏返回为真时，才会考虑该任务。在 UP 情况下，can_schedule 宏返回恒为真.
⑨如果循环结束后，得到 c 的值为 0。说明运行队列中的所有进程的 goodness 值都为 0。goodness 的值为 0,意味着进程已经用完它的时间片，或者它已经明确说明要释放 CPU。在这种情况下，schedule 要重新计算进程的 counter；新 counter 的值是原来值的一半加上进程

的静态优先级（priortiy），除非进程已经释放 CPU，否则原来 counter 的值为 0。因此，schedule 通常只是把 counter 初始化为静态优先级。（中断处理程序和由另一个处理器引起的分支在 schedule 搜寻 goodness 最大值时都将增加此循环中的计数器，因此由于这个原因计数器可能不会为 0。显然，这很罕见。）在 counter 的值计算完成后，重新开始执行这个循环，找具有最大 goodness 的任务。

⑩如果 schedule 已经选择了一个不同于前面正在执行的进程来调度，那么就必须挂起原来的进程并允许新的进程运行。这时调用 switch_to 来进行切换。

从调度的角度，Linux 把进程分成 140 个优先等级，其中 0 级到 99 级是分给实时进程的，100 级到 139 级是分给非实时进程的。每个优先等级都有一个运行对列，这样就有 140 个运行队列。级数越小优先度越高。调度程序从 0 级到 139 级依次询问每个运行队列是否有可执行进程。询问的方法是通过访问一个 bitmap，这个位图共有 160bits，前 140 位与 140 个运行队列一一对应，后 20 位空闲。在每个运行队列里又把进程分成两组队列，active 队列和 expired 队列。active 队列相当于前面说的那个圆盘，里面的进程是被分配了时间片的。expired 队列相当于圆盘外的“替补席”。一般来说，active 队列里的进程用完其时间片后，就被送到 expired 队列里（也可能由于要和某个外设同步而被直接送到“休息室”--由可执行状态变成等待状态）。当 active 队列里的进程都被送出去的时候，就把 active 队列和 expired 队列互换一下，原来的 active 队列变成 expired 队列，原来的 expired 队列变成 active 队列。这就是 Linux 进程调度的基本思路。

Linux 进程的优先级有两种，静态优先级和动态优先级。

静态优先级是初始优先级，它决定时间片的大小。

时间片大小 = （140 - 静态优先级） X 20 ;(静态优先级

时间片大小 = （140 - 静态优先级） X 5 ;(静态优先级 >= 120)

和静态优先级有关的一个变量叫做 nice。进程外部可以通过调整 nice 值改变静态优先级。

静态优先级 = 120 - nice 值

动态优先级是在调度的过程中不断变化的。它决定该进程被送到 140 个运行队列中的哪一个。

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

和动态优先级有关的一个变量叫做 bonus。动态优先级所谓的“动态”就体现在 bonus 的变化上。bonus 的值和进程的睡眠时间反相关，睡眠时间越长 bonus 值越大，睡眠时间越短 bonus 值越小。bonus 的取值范围是 0-10。另外 bonus 还有一个作用就是可以作为区分交互进程和批处理进程的标准。

当 bonus - 5 >=静态优先级 / 4- 28 时，该进程被看作交互进程。

当 bonus - 5