**一、进程优先级**

进程的优先级有2种度量方法，一种是nice值，一种是实时优先级（RTPRIO）。

nice值的范围是-20～+19，值越大优先级越低，也就是说nice值为-20的进程优先级最大。

实时优先级（RTPRIO）的范围是0～99，与nice值的定义相反，实时优先级是值越大优先级越高。

实时进程都是一些对响应时间要求比较高的进程，因此系统中有实时优先级高的进程处于运行队列的话，它们会抢占一般的进程的运行时间。

  进程的2种优先级会让人不好理解，到底哪个优先级更优先？一个进程同时有2种优先级怎么办？

对于第一个问题，到底哪个优先级更优先？答案是实时优先级高于nice值，在内核中，实时优先级的范围是 0～MAX\_RT\_PRIO-1。第二个但是进程不会同时拥有两个优先级。

MAX\_RT\_PRIO的定义参见 include/linux/sched.h

1611 #define MAX\_USER\_RT\_PRIO 100

1612 #define MAX\_RT\_PRIO MAX\_USER\_RT\_PRIO

nice值在内核中的范围是 MAX\_RT\_PRIO～MAX\_RT\_PRIO+40 即 MAX\_RT\_PRIO～MAX\_PRIO

1614 #define MAX\_PRIO (MAX\_RT\_PRIO + 40)

查看进程优先级：

$ ps -eo state,uid,pid,ppid,rtprio,ni,time,comm

S UID PID PPID RTPRIO NI TIME COMMAND

S 0 1 0 - 0 00:00:00 systemd

S 0 2 0 - 0 00:00:00 kthreadd

S 0 3 2 - 0 00:00:00 ksoftirqd/0

S 0 6 2 99 - 00:00:00 migration/0

S 0 7 2 99 - 00:00:00 watchdog/0

S 0 8 2 99 - 00:00:00 migration/1

S 0 10 2 - 0 00:00:00 ksoftirqd/1

S 0 12 2 99 - 00:00:00 watchdog/1

S 0 13 2 99 - 00:00:00 migration/2

S 0 15 2 - 0 00:00:00 ksoftirqd/2

S 0 16 2 99 - 00:00:00 watchdog/2

S 0 17 2 99 - 00:00:00 migration/3

S 0 19 2 - 0 00:00:00 ksoftirqd/3

S 0 20 2 99 - 00:00:00 watchdog/3

S 0 21 2 - - 20 00:00:00 cpuset

S 0 22 2 - - 20 00:00:00 khelper

**二、时间片**

对于调度程序来说，并不是运行一次就结束了，还必须知道间隔多久进行下次调度。需要有时间片的概念。时间片是一个数值，表示一个进程被抢占前能持续运行的时间。也可以认为是进程在下次调度发生前运行的时间(除非进程主动放弃CPU，或者有实时进程来抢占CPU)。时间片的大小设置并不简单，设大了，系统响应变慢(调度周期长)；设小了，进程频繁切换带来的处理器消耗。默认的时间片一般是10ms

**三、基于时间片+优先级的调度**

实现原理

1）确定每个进程能占用多少CPU时间(这里确定CPU时间的算法有很多，根据不同的需求会不一样)

2）占用CPU时间多的先运行

3）运行完后，扣除运行进程的CPU时间，再回到 1）

**说明例子：**

假设系统中只有3个进程ProcessA(NI=+10)，ProcessB(NI=0)，ProcessC(NI=-10)，NI表示进程的nice值，时间片=10ms

1) 调度前，把进程优先级按一定的权重映射成时间片(这里假设优先级高一级相当于多5msCPU时间)。

    假设ProcessA分配了一个时间片10ms，那么ProcessB的优先级比ProcessA高10(nice值越小优先级越高)，ProcessB应该分配10\*5+10=60ms，以此类推，ProcessC分配20\*5+10=110ms

2) 开始调度时，优先调度分配CPU时间多的进程。

由于ProcessA(10ms),ProcessB(60ms),ProcessC(110ms)。显然先调度ProcessC

3) 10ms(一个时间片)后，再次调度时，ProcessA(10ms),ProcessB(60ms),ProcessC(100ms)。ProcessC刚运行了10ms，所

以变成100ms。此时仍然先调度ProcessC

4) 再调度4次后(4个时间片)，ProcessA(10ms),ProcessB(60ms),ProcessC(60ms)。

此时ProcessB和ProcessC的CPU时间一样，这时得看ProcessB和ProcessC谁在CPU运行队列的前面，假设ProcessB在前面，则调度ProcessB

5) 10ms(一个时间片)后，ProcessA(10ms),ProcessB(50ms),ProcessC(60ms)。再次调度ProcessC

6) ProcessB和ProcessC交替运行，直至ProcessA(10ms)，ProcessB(10ms) ，ProcessC(10ms)。

    这时得看ProcessA，ProcessB，ProcessC谁在CPU运行队列的前面就先调度谁。这里假设调度ProcessA

7) 10ms(一个时间片)后，ProcessA(时间片用完后退出),ProcessB(10ms),ProcessC(10ms)。

8) 再过2个时间片，ProcessB和ProcessC也运行完退出。

**四、调度实现**

2.6.23内核以后，采用了“完全公平调度算法”，简称CFS。CFS算法在分配每个进程的CPU时间时，不是分配给它们一个绝对的CPU时间，而是根据进程的优先级分配给它们一个占用CPU时间的百分比。

比如：ProcessA(NI=1)，ProcessB(NI=3)，ProcessC(NI=6)，在CFS算法中，分别占用CPU的百分比为：ProcessA(10%)，ProcessB(30%)，ProcessC(60%)

因为总共是100%，ProcessB的优先级是ProcessA的3倍，ProcessC的优先级是ProcessA的6倍。

  Linux上的CFS算法主要有以下步骤：

(还是以ProcessA(10%)，ProcessB(30%)，ProcessC(60%)为例)

1)计算每个进程的vruntime，通过update\_curr()函数更新进程的vruntime。

2)选择具有最小vruntime的进程投入运行。

3)进程运行完后，更新进程的vruntime，转入步骤2)

**五、与调度相关的调用**

调度相关的系统调用主要有2类：

1) 与调度策略和进程优先级相关 (各种参数，优先级，时间片)

|  |  |
| --- | --- |
| **系统调用** | **描述** |
| nice() | 设置进程的nice值 |
| sched\_setscheduler() | 设置进程的调度策略，即设置进程采取何种调度算法 |
| sched\_getscheduler() | 获取进程的调度算法 |
| sched\_setparam() | 设置进程的实时优先级 |
| sched\_getparam() | 获取进程的实时优先级 |
| sched\_get\_priority\_max() | 获取实时优先级的最大值，由于用户权限的问题，非root用户并不能设置实时优先级为99 |
| sched\_get\_priority\_min() | 获取实时优先级的最小值，理由与上面类似 |
| sched\_rr\_get\_interval() | 获取进程的时间片 |

2) 与处理器相关

|  |  |
| --- | --- |
| **系统调用** | **描述** |
| sched\_setaffinity() | 设置进程的处理亲和力，其实就是保存在task\_struct中的cpu\_allowed这个掩码标志。  该掩码的每一位对应一个系统中可用的处理器，默认所有位都被设置，即该进程可以再  系统中所有处理器上执行。用户可以通过此函数设置不同的掩码，使得进程只能在系统中  某一个或某几个处理器上运行。 |
| sched\_getaffinity() | 获取进程的对处理机的附着度 |
| sched\_yield() | 暂时让出处理器 |