**主要内容：**

* 什么是调度
* 调度实现原理
* Linux上调度实现的方法
* 调度相关的系统调用

### ****1. 什么是调度****

管理计算机上同时运行的各个任务（也就是进程）的程序。

这个管理程序就是调度程序，它的功能说起来很简单：

决定哪些进程运行，哪些进程等待

决定每个进程运行多长时间

### ****2. 调度实现原理****

#### **2.1 关于进程的优先级**

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。

nice值的范围是-20～+19，值越大优先级越低

实时优先级的范围是0～99，实时优先级是值越大优先级越高。

进程的2种优先级到底哪个优先级更优先？

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

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

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

一个进程不可能有2种优先级

我们可以通过以下命令查看进程的实时优先级和Nice值：(其中RTPRIO是实时优先级，NI是Nice值)

$ 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

#### **2.2 关于时间片**

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)。

时间片的大小默认的时间片一般是10ms

#### **2.3 调度实现原理（注：基于优先级和时间片）**

下面举个直观的例子来说明：

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)。
2. 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
3. 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
4. 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
5. 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
6. 10ms(一个时间片)后，ProcessA(10ms),ProcessB(50ms),ProcessC(60ms)。再次调度ProcessC
7. ProcessB和ProcessC交替运行，直至ProcessA(10ms),ProcessB(10ms),ProcessC(10ms)。
8. 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
9. 10ms(一个时间片)后，ProcessA(时间片用完后退出),ProcessB(10ms),ProcessC(10ms)。
10. 再过2个时间片，ProcessB和ProcessC也运行完退出。

### ****3. Linux上调度的方法****

在2.6.23内核以后，采用了“完全公平调度算法”，简称CFS。

CFS算法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%)

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(注1)，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) （注3）

**注1.** 这里的vruntime是进程虚拟运行的时间的总和。 **vruntime**并不是实际的运行时间，它是**实际运行时间进行加权运算**后的结果。

比如上面3个进程中ProcessA(10%)只分配了CPU总的处理时间的10%，那么ProcessA运行10ms的话，它的vruntime会增加100ms。

以此类推，ProcessB运行10ms的话，它的vruntime会增加(100/3)ms,ProcessC运行10ms的话，它的vruntime会增加(100/6)ms。

**注3.**Linux为了能快速的找到具有最小vruntime，将所有的进程的存储在一个红黑树中。这样树的最左边的叶子节点就是具有最小vruntime的进程，新的进程加入或有旧的进程退出时都会更新这棵树。

其实Linux上的调度器是以模块方式提供的，每个调度器有不同的优先级，所以可以同时存在多种调度算法。

每个进程可以选择自己的调度器，Linux调度时，首先按调度器的优先级选择一个调度器，再选择这个调度器下的进程。

CFS使用调度器实体结构（定义在文件vlinux/ sched.h＞的struct\_sched\_entity中）来追踪进程运行记账:

struct sched\_entity {

struct load\_weight load；

struct rb\_\_node run\_node；

struct list\_head group\_node；

unsigned int on\_rq；

u64 exec\_start；

u64 sum\_exec\_runtime；

u64 vruntime；

U64 prev\_sum\_exec\_runt ime;

u64 last\_wakeup；

U64 avg\_overlap；

U64 nr\_migrations；

u64 start\_runtime；.

u64 avg\_wakeup；

/\* 这里省略了很多统计变S：，只有在设置了 CONFIG\_SCHEDSTATS时才启用这些变量\*/

}

调度器实体结构作为一个名为se的成员变量，嵌入在进程描述符struct task\_struct内。

进程调度的主要入口点是函数schedule(),它定义在文件kemel/sched.c中。该函数中唯一重要的事情是，它会调用pick\_next\_task()(也定义在文件kernel/sched.c中)。pick\_ next\_task()会以优先级为序，从高到低，依次检査毎一个调度类，并且从最高优先级的调度类中，选择最高优先级的进程

### ****4. 调度相关的系统调用****

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 (就是上面的提到的各种参数，优先级，时间片等等) - 下表中的前8个

2) 与处理器相关 - 下表中的最后3个

|  |  |
| --- | --- |
| **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 |
| 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() | 获取进程的处理亲和力 |
| sched\_yield() | 暂时让出处理器 |

**与调度策略和优先级相关的系统调用**

sched\_setscheduler()和sched\_getscheduler()分别用于设置和获取进程的调度策略和实时优先级。

**放弃处理器时间**

Linux通过sched\_yield()系统调用，提供了一种让进程显式地将处理器时间让给其他等待执行进程的机制。

1. **休眠和唤醒**

休眠(被阻塞)的进程处于一个特殊的不可执行状态。进程把自己标记成休眠状态，从可执行红黑树中移出，放入等待队列，然后调用schedule()选择和执行一个其他进程。唤醒的过程刚好相反：进程被设置为可执行状态，然后再从等待队列中移到可执行红黑树中。

在第3章里曾经讨论过，休眠有两种相关的进程状态:TASK\_INTERRUPTIBLE和TASK\_ UNINTERRUPTIBLEO它们的唯一区别是处于TASK\_UNINTERRUPTIBLE的进程会忽略信号， 而处于TASK\_INTERRUPTIBLE状态的进程如果接收到一个信号，会被提前唤醒并响应该信号。 两种状态的进程位于同一个等待队列上，等待某些事件，不能够运行。