2.5 示例 Linux 的进程调度算法

1．Linux 进程调度方式：

Linux 系统采用抢占调度方式。Linux2.6 内核是抢占式的，这意味着进程无论是处于内核态还是用户态，都可能被抢占。

Linux 的调度基于分时技术（time-sharing）。对于优先级相同进程进程采用时间片轮转法。

根据进程的优先级对它们进行分类。进程的优先级是动态的。

2．Linux 进程调度策略

task_struct 中与进程调度相关的一些变量有：unsigned long policy。

在 Linux 内核 include/linux/sched.h 文件中定义：

```
#define SCHED_NORMAL 0
#define SCHED_FIFO        1
#define SCHED_RR          2
#define SCHED_BATCH     3
/* SCHED_ISO: reserved but not implemented yet */
#define SCHED_IDLE        5  //最不重要的进程
/* Can be ORed in to make sure the process is reverted back to SCHED_NORMAL on fork */
#define SCHED_RESET_ON_FORK     0x40000000
```

Linux 的进程分为普通进程和实时进程，实时进程的优先级高于普通进程。Linux 进程的常用策略 policy 有:

符号常量	意义
SCHED_NORMAL	普通进程的时间片轮转算法
SCHED_FIFO	实时进程的先进先出算法
SCHED_RR	实时进程的时间片轮转算法
SCHED_BATCH	后台处理进程

普通进程按照 SCHED_NORMAL 调度策略进行进程调度；实时进程按照 SCHED_FIFO 或 SCHED_RR 策略进行调度。

SCHED_BATCH 是 2.6 新加入的调度策略，这种类型的进程一般都是后台处理进程，总是倾向于跑完自己的时间片，没有交互性。所以对于这种调度策略的进程，调度器一般给的优先级比较低，这样系统就能在没什么事情做的时候运行这些进程，而一旦有交互性的进程需要运行，则立刻切换到交互性的进程，从用户的角度来看，系统的响应性/交互性就很好。

3．进程的调度算法

调度程序源代码在 kernel/sched.c 文件中。

在 Linux 2.4 版本及以前内核版本中，调度算法采用传统的 UNIX 调度算法：遍历进程可运行队列，算法时间为 O(n)；采用时间片和动态优先级

在 Linux 2.6.1-2.6.22 内核中调度算法使用优先级队列调度的 O(1)算法：优先级从 0 到 139 分成 140 个优先级队列；使用 active 和 expire 两个队列，按照优先级调度，算法时间为 O(1)。

在 Linux 2.6.23 版至目前的内核版本中增加了一种称为完全公平调度算法（Completely Fair Scheduler， CFS） 。对于非实时进程使用 CFS (完全公平调度器）进程调度器，使用红黑树选取下一个被调度进程，O(lg N)；对于实时进程使用优先级队列调度。

4．Linux 2.6 O(1)进程调度算法

Linux 2.6 内核 O(1)调度算法是抢占的、基于优先级的算法。进程设置 140 个优先级，实时进程优先级为 0-99，普通进程优先级 100-139 的数，这两个范围映射到全局优先级，其中数值越小表明优先级越高，0 为最高优先权，139 为最低优先权。

优先级分为静态优先级和动态优先级。调度程序根据动态优先级来选择新进程运行。静态优先权本质上决定了进程的基本时间片，即进程用完了以前的时间片时，系统分配给进程的时间片长度。静态优先权和基本时间片（base time quantum）的关系用下列公式确定：

base time quantum （ms）：

- （140-static_priority）*20 if static_priority <120
- （140-static_priority）*5 if static_priority >=120

task_struct 中与调度有关的字段如下：

- sleep_avg 字段，用于存储进程的平均等待时间（nanosecond），它反映交互式进程优先与分时系统的公平共享，值越大，计算出来的进程优先级也越高
- run_list 字段，串连在优先级队列中，优先级数组 prio_array 中按顺序排列了各个优先级下的所有进程，调度器在 prio_arrary 中找到相应的 run_list，从而找到其宿主结构 task_struct
- time_slice 字段，进程的运行时间片剩余大小；进程的默认时间片与进程的静态优先级相关；进程创建时，与父进程平分时间片；运行过程中递减，一旦归零，则重置时间片，并请求调度；递减和重置在时钟中断中进行（scheduler_tick()）；进程退出时，如果自身并未被重新分配时间片，则将自己剩余的时间片返还给父进程。
- static_prio 字段，静态优先级，与 2.4 版本中的 nice 值意义相同，但取值区间不同，是用户可影响的优先级；通过 set_user_nice()来改变；static_prio= MAX_RT_PRIO + nice + 20 （其中，MAX_RT_PRIO 定义为 100）；进程初始时间片的大小仅决定于进程的静态优先级；核心将 100~139 的优先级映射到 200ms~10ms 的时间片上；优先级数值越大，优先级越低，分配的时间片越少；实时进程的 static_prio 不参与优先级 prio 的计算

- prio 字段，动态优先级，相当于 2.4 中 goodness() 的计算结果，在 0~MAX_PRIO-1 之间取值（MAX_PRIO 定义为 140），其中：

  0~MAX_RT_PRIO-1 （MAX_RT_PRIO 定义为 100）属于实时进程范围；

  MAX_RT_PRIO~MAX_PRIO-1 属于非实时进程。数值越大，表示进程优先级越小。

2.6 中，动态优先级不再统一在调度器中计算和比较，而是独立计算，并存储在进程的 task_struct 中，再通过描述的 priority_array 结构自动排序。

普通进程 prio = max (100, min (static priority - bonus + 5, 139))

Bonus 是范围从 0 到 10 的值，bonus 的值小于 5 表示降低动态优先权以示惩罚，bonus 的值大于 5 表示增加动态优先权以示额外奖赏。Bonus 的值依赖于进程过去的情况，说得更准确一些是与进程的平均睡眠时间相关。

prio 的计算和很多因素相关。

- unsigned long rt_priority 字段，实时进程的优先级；一经设定在运行时不变，作为其动态优先级。

sys_sched_setschedule()

- prio_array_t *array 字段，记录当前 CPU 活动的就绪队列；以优先级为序组成数组

runqueue 结构（kernel/sched.c）

runqueue 结构是 Linux2.6 调度程序最重要的数据结构。系统中的每个 CPU 都有它自己的运行队列，所有的 runqueue 结构存放在 runqueues 每 CPU（ per-CPU）变量中。
系统中的每个可运行进程属于且只属于一个运行队列。只要可运行进程保持在同一个运行队列中，它就只可能在拥有该运行队列的 CPU 上执行。但是，可运行进程会从一个运行队列迁移到另一个运行队列。
runqueue 结构的字段及含义如下表

Type	**Name**	**Description**
spinlock_t	lock	保护进程链表的自旋锁
unsigned long	nr_running	运行队列链表中可运行进程的数量
unsigned long	cpu_load	基于运行队列中进程的平均数量的 CPU 负载因子
unsigned long	nr_switches	CPU 执行进程切换的次数
unsigned long	nr_uninterruptible	先前在运行队列链表中而现在睡眠在 TASK_UNINTERRUPTIBLE 状态的进程的数量（对所有运行队列来说，这些字段的总数才是有意义的）
unsigned long	expired_timestamp	过期队列中最老的进程被插入队列的时间。
unsigned long long	timestamp_last_tick	最近一次定时器中断的时间戳的值
task_t *	curr	当前正在运行进程的进程描述符指针（对本地 CPU，它与 current 相同）
task_t *	idle	当前 CPU（this CPU）上交换进程的进程描述符指针。
struct mm_struct *	prev_mm	在进程切换期间用来存放被替换进程的内存描述符的地址
prio_array_t *	active	指向活动进程链表的指针
prio_array_t *	expired	指向过期进程链表的指针
prio_array_t [2]	arrays	活动和过期进程的两个集合
int	best_expired_prio	过期进程中静态优先权最高的进程（权值最小）。
atomic_t	nr_iowait	先前在运行队列的链表中而现在正等待磁盘 I/O 操作结束的进程的数量。
struct sched_domain *	sd	指向当前 CPU 的基本调度域

int	active_balance	如果要把一些进程从本地运行队列迁移到另外的运行队列（平衡运行队列），就设置这个标志。
int	push_cpu	未使用
task_t *	migration_thread	迁移内核线程的进程描述符指针。
struct list_head	migration_queue	从运行队列中被删除的进程的链表

其中 runqueue 结构中的字段：

prio_array_t *active, *expired, arrays[2]

每个 CPU 均有两个具有优先级的队列，按时间片是否用完分为“活动队列”（active 指针所指）和“过期队列”（expired 指针所指）。active 指向时间片没用完、当前可被调度的就绪进程，expired 指向时间片已用完的就绪进程。

每一类队列用一个 struct prio_array 表示（优先级排序数组）；一个任务的时间片用完后，它会被转移到“过期”的队列中；在该队列中，任务仍然是按照优先级排好序的当活动队列中的任务均被执行完时，就交换两个指针。

每一类就绪进程都用一个 struct prio_array 的结构表示

```
struct prio_array{
    unsigned int nr_active; /*本进程组中的进程数*/
    unsigned long bitmap[BITMAP_SIZE];/*加速 HASH 表访问的位图快速定位第一个非空的就绪进程链表*/
    struct list_headqueue[MAX_PRIO];/*以优先级为索引的 HASH 表*/
}
```

进程调度由 schedule()函数实现。首先，schedule()利用下面的代码定位优先级最高的就绪进程：

```
prey=current;
array=rq->active;
idx=sched_find_first_bit(array->bitmap):
queue= array- >queue+idx;
next =list_entry(queue->next，struct task_struct，run_list);
```

schedule()通过调用 sched_find_first_bit()函数找到当前 CPU 就绪进程队列 runqueue 的 active 进程数组中第一个非空的就绪进程链表。这个链表中的进程具有最高的优先级，schedule()选择链表中的第一个进程作为调度器下一时刻将要运行的进程。

如果 prev(当前进程)和 next(将要运行的进程)不是同一个进程，schedule()调用 context_switch()将 CPU 切换到 next 进程运行。

调度程序依靠几个函数来完成调度工作，其中最重要的函数有：

scheduler_tick( )，维持当前最新的 time_slice 计数器

try_to_wake_up( )，唤醒睡眠进程

recalc_task_prio( )，更新进程的动态优先权

schedule( )，选择要被执行的新进程

load_balance()，维持多处理器系统中运行队列的平衡。