第15章 进程调度

实验目的

- 深入理解进程调度的基本原理
- 了解Linux2.4调度算法及其不足
- 掌握Linux2.6调度算法的原理与实现
- 根据要求能对Linux2.6调度算法进行修改

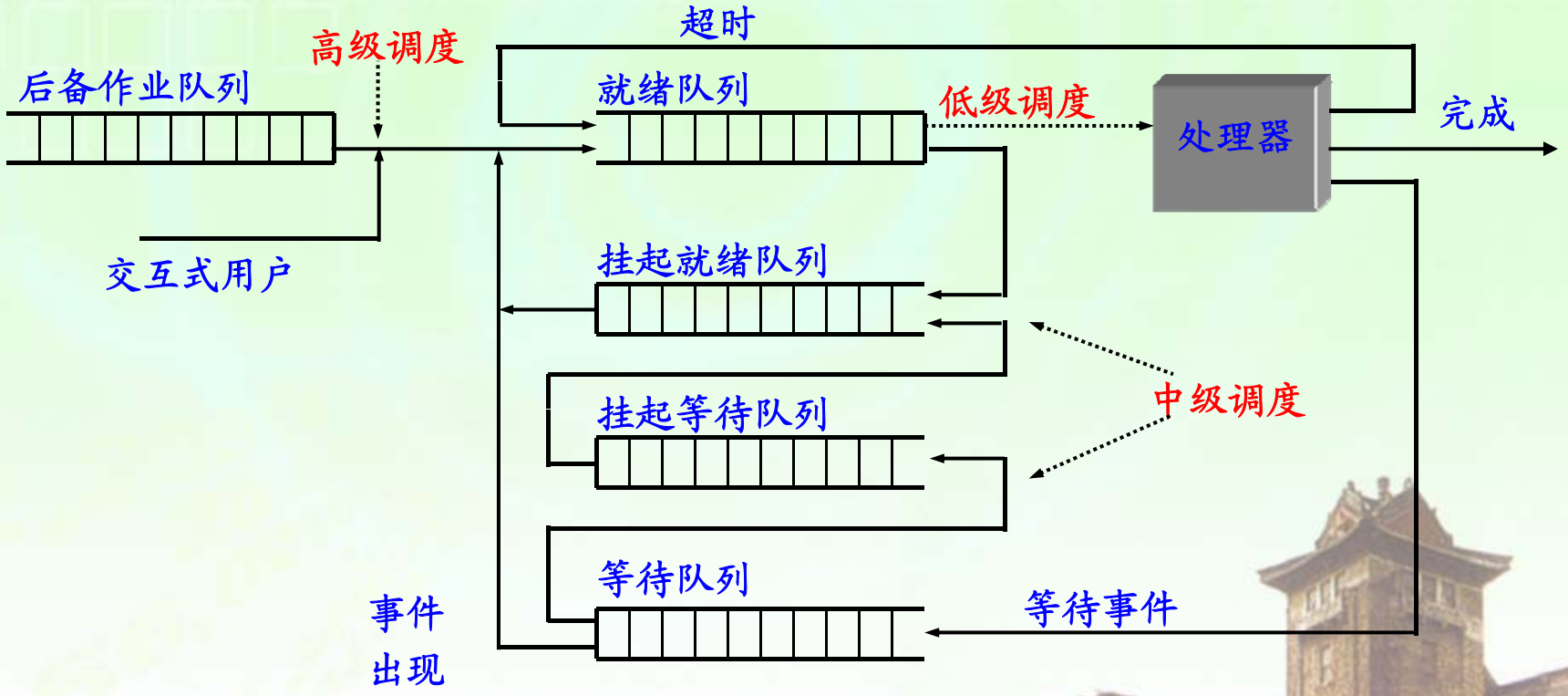
主要内容

- 背景知识
  - 调度策略和调度机制
  - Linux 2.4的进程调度机制
  - Linux 2.6的进程调度机制
- 实验内容
  - 将Linux 2.6调度算法修改成随机调度算法

调度分类

- 高级调度
  - 又称作业调度、长程调度
  - 选出若干作业进入主存，分配所需资源，创建对应作业的用户进程后
  - 控制多道程序的道数，作业数越多，每个作业获得的CPU时间越少
- 中级调度(Medium Level Scheduling)
  - 根据主存资源量决定主存中所能容纳的进程数，以及进程的当前状态来决定辅存和主存中的进程的对换
  - 短期平滑和调整系统负荷，充分提高主存利用率和系统吞吐率
- 低级调度(Low Level Scheduling)
  - 又称进程调度/线程调度、短程调度
  - 从就绪队列中选择进程/内核级线程，并将处理器出让给它使用
  - 操作系统最为核心的部分，执行十分频繁，常驻主存

处理器的三级调度模型


超时
高级调度
后备作业队列
就绪队列
低级调度
处理器
完成
交互式用户
挂起就绪队列
中级调度
挂起等待队列
等待队列
事件
出现
等待事件

进程调度概述

- 目标
  - 最大限度地利用处理器时间
- 基本任务
  - 选择下一个要运行的进程
- 核心问题
  - 在可运行态进程之间分配有限的处理器时间，保证各进程公平、有效地使用处理器时间资源
- 难点
  - 响应时间vs.系统开销
  - 不同类型的进程有不同的调度需求
- 调度器的基本要求
  - 减少调度器自身开销，以增加花在执行程序上的时间
  - 对交互式应用有良好的响应速度

多任务系统分类

- 非抢占式多任务(cooperative multitasking)
  - 工作模式：让步（yielding）
    - ✓除非进程主动停止运行，否则一直执行
  - 特点
    - ✓无法预料进程独占处理器的时间，易导致系统崩溃
- 抢占式多任务(preemptive multitasking)
  - 工作模式：抢占（preemption）
    - ✓进程在被抢占前能够运行的时间是预先设置好的，称之为时间片（timeslice）
  - 特点
    - ✓可避免个别进程独占系统资源
    - ✓调度策略？

进程的分类

- 第一种分类
  - I/O-bound
    - ✓I/O操作频繁
    - ✓通常会花费很多时间等待I/O操作的完成
    - ✓不需要太长的时间片
  - CPU-bound
    - ✓计算密集型
    - ✓需要大量的CPU时间进行运算

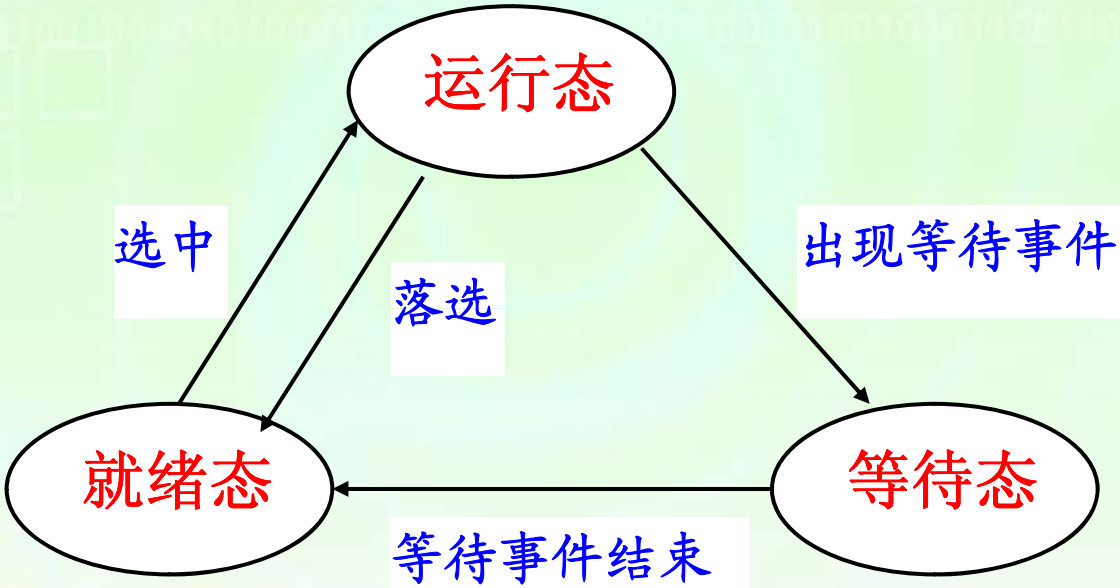
进程的分类

- 第二种分类方法
  - 交互式进程（interactive process）
    - ✓ 需要经常与用户交互，因此要花很多时间等待用户输入操作
    - ✓ 响应时间要快，平均延迟要低于50~150ms
    - ✓ 典型的交互式程序：shell、文本编辑程序、图形应用程序等
  - 批处理进程（batch process）
    - ✓ 不必与用户交互，通常在后台运行
    - ✓ 不必很快响应
    - ✓ 典型的批处理程序：编译程序、科学计算
  - 实时进程（real-time process）
    - ✓ 有实时需求，不应被低优先级的进程阻塞
    - ✓ 响应时间要短、要稳定
    - ✓ 典型的实时进程：视频/音频、机械控制等

调度策略

- 决定什么时候以怎样的方式选择一个新进程运行的一组规则
- 难点
  - 进程响应时间尽可能快（切换时机）
  - 后台作业吞吐量尽可能高（降低切换开销）
  - 协调不同进程的调度顺序（优先级定义）
  - 尽可能避免进程的饥饿现象（动态优先级更新）

进程状态转换模型



Linux进程调度的基本机制

- 调度方法
  - 基于分时技术的抢占调度方式
    - CPU的时间被划分成“时间片”，给每个可运行进程分配一片
    - 分时依赖于时钟中断，对进程透明
- 调度策略
  - 进程分类
    - 普通进程
    - 实时进程
  - 优先级
    - 静态优先级
    - 动态优先级

主要内容

- 背景知识
  - 调度策略和调度机制
  - **Linux 2.4的进程调度机制**
  - Linux 2.6的进程调度机制
- 实验内容
  - 将Linux 2.6调度算法修改成随机调度算法

Linux 2.4的进程调度

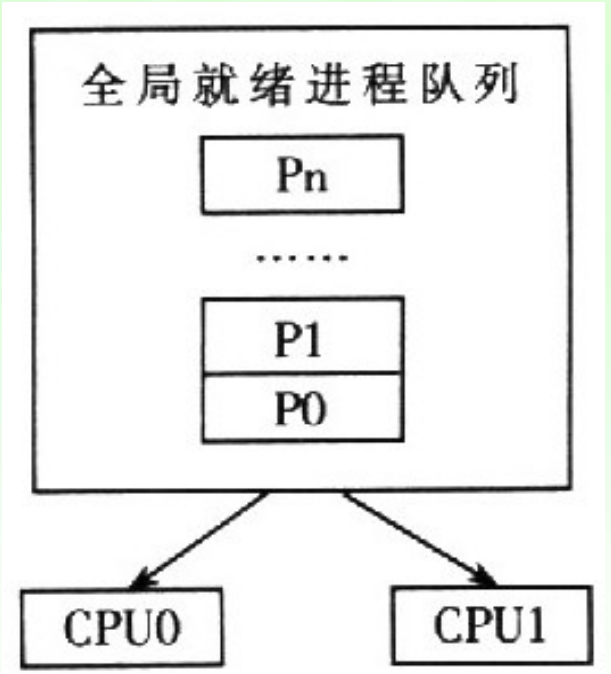
- 进程调度结构
- 调度相关的基本数据
- 调度策略
- 调度时机
- **schedule()分析**

Linux 2.4中进程管理相关数据结构

- **task[NR_TASKS]**
  - 所有进程（包括0号进程）被组织到以**init_task**为表头的双向链表([include/linux/sched.h]SET_LINKS()宏)，该链表是全系统唯一
- **init_tasks[NR_CPUS]**
  - 所有CPU的0号进程的数组，构成一个链表，它实际上是task[NR_TASKS]链表的子链表
  - SMP系统中，每个CPU都分别对应了一个**idle_task**，其task_struct指针被组织到init_tasks[NR_CPUS]数组中
  - 调度器通过idle_task(cpu)宏来访问这些“idle”进程
- **runqueue_head**
  - 所有就绪进程（状态为可运行的进程）构成一个链表，表头是runqueue_head

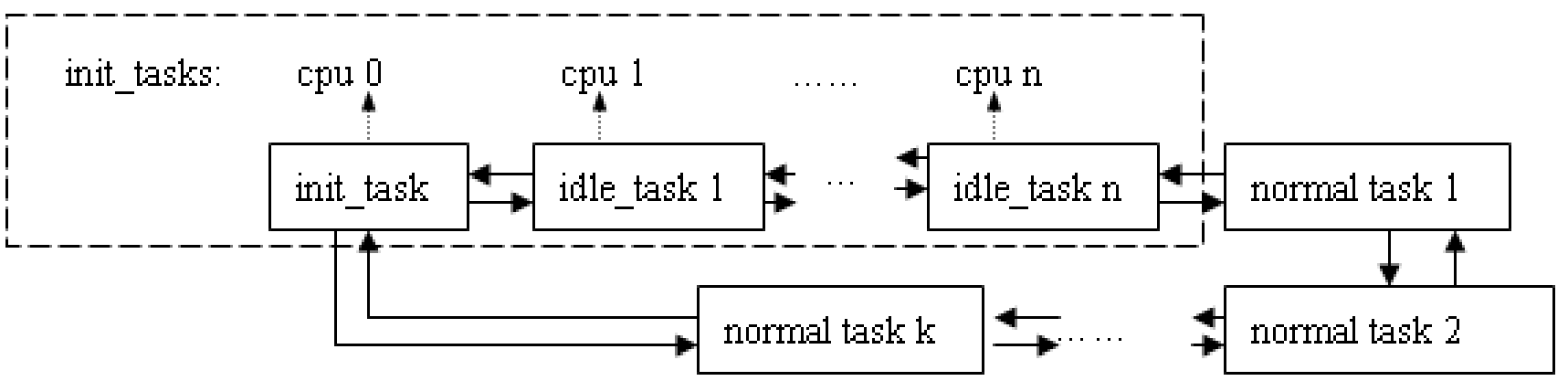
Linux 2.4的调度结构

- 所有就绪进程存在一个以**runqueue_head**为表头的全局进程队列中（当前正在运行的进程也在其中，但**idle_task**除外），调度器从中选取最适合调度的进程投入运行
- 整个队列由一个读/写自旋锁保护着，这样多个处理器可以并行访问，但同时提供写操作的互斥访问。
- 时间片重算算法是在所有的进程都用尽它们的时间片以后才重新计算



Linux 2.4的init tasks结构

- 表头是启动CPU（BSP）的0号进程，即init_task
- 调度器并不直接使用init_task为表头的进程链表，而仅使用其中的“idle_task”。该进程在引导完系统后即处于cpu_idle()循环中
- 新进程总是添加到init_task的左端，即prev端



idle进程

- 系统最初的引导进程（init_task）在引导结束后即成为cpu 0上的idle进程
- 在每个cpu上都有一个idle进程，这些进程登记在init_tasks[]数组中，并可用idle_task()宏访问
- idle进程不进入就绪队列，系统稳定后，仅当就绪队列为空的时候idle进程才会被调度到

cpu_idle()

- init_task的nice值被设为20（最低优先级），counter为-100（无意义的足够小），然后cpu_idle()进入无限循环
- 初始化过程中第一次执行cpu_idle()，need_resched为1，所以直接启动schedule()进行第一次调度
- schedule()会清掉need_resched位，因此，之后本循环都将执行idle()函数，直至need_resched再被设置为非0

```
while (1) {
    void (*idle)(void) = pm_idle;
    if (!idle)
        idle = default_idle;
    while (!current->need_resched)
    idle();
    schedule();
    check_pgt_cache();
}
```

Linux 2.4的schedule data结构

❖结构定义

```
static union {
  struct schedule_data {
    //此CPU上的当前进程，通常用cpu_curr(cpu)宏来访问
    struct task_struct * curr;
    //此CPU上次进程切换的时间，通常用last_schedule(cpu)宏来访问
    cycles_t last_schedule;
  } schedule_data;
  char __pad [SMP_CACHE_BYTES];
}
```

❖功能

- 访问到某CPU上运行的进程
- 所有CPU被组织到以schedule_data为元素的数组aligned_data [NR_CPUS]之中，每个元素代表一个CPU

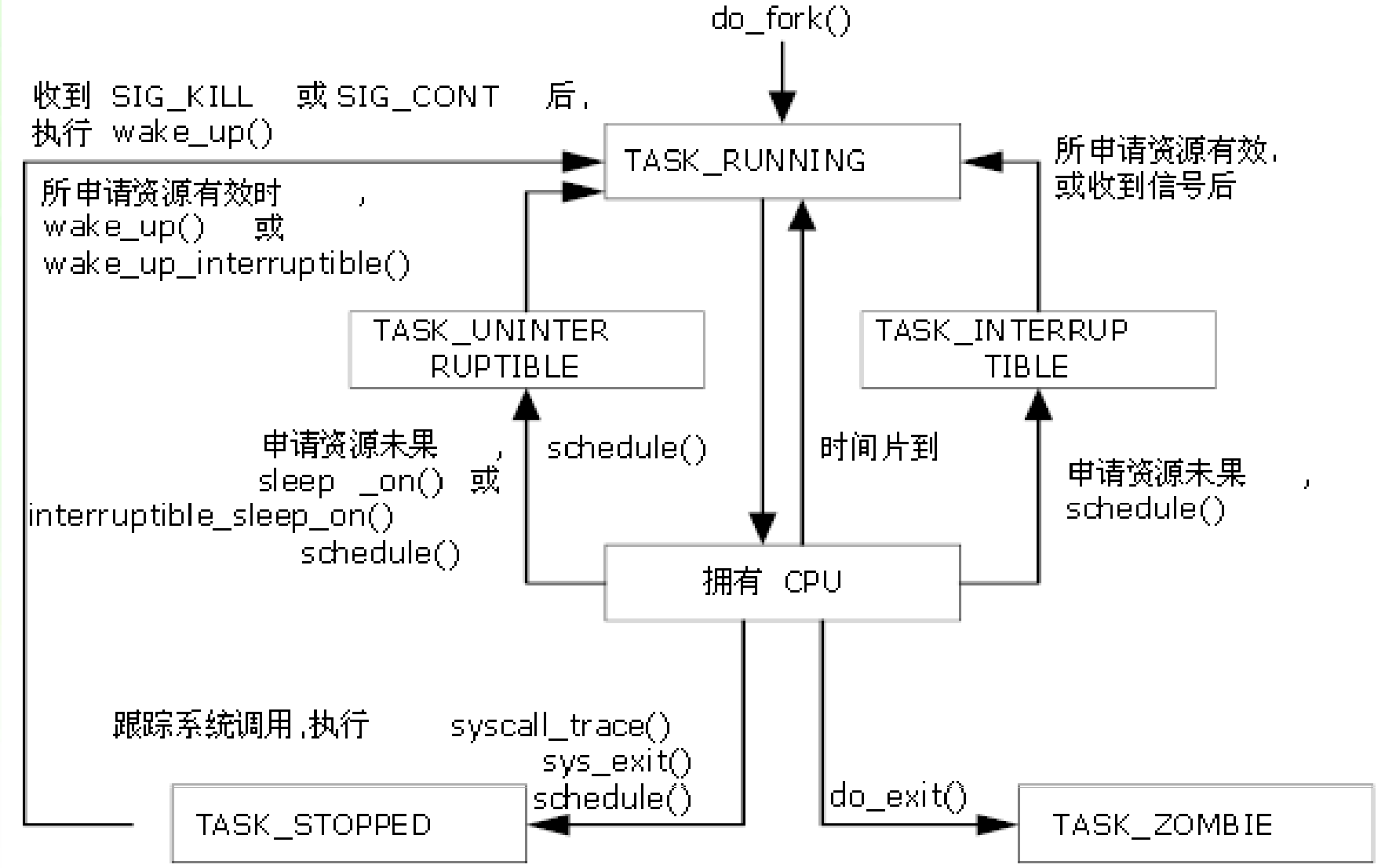
Linux 2.4进程描述符中与调度相关的域

变量名	说明
state	进程的当前状态
need_resched	布尔值，表示该进程是否需要申请调度
policy	进程的调度类型
rt_priority	进程的实时优先级
nice	用户可控制的进程优先级
priority	静态优先级，实时进程忽略该成员
count	当前时间片内该进程的剩余运行时间
cpus_allowed	以位向量形式表示可执行该进程的CPU
cpus_runnable	以位向量形式表示当前运行该进程的CPU
processor	本进程当前（或最近）所在CPU编号
thread	用于保存进程执行环境（各寄存器的值及IO操作许可权映射表）

Linux 2.4中的进程状态

- 域成员名：**state**
- 状态分类
  - 运行态/就绪态
    - ✓ **TASK_RUNNING：正在运行，或已处于就绪只等待CPU调度**
  - 被挂起状态
    - ✓ **TASK_INTERRUPTIBLE：可被信号或中断唤醒而进入就绪队列**
    - ✓ **TASK_UNINTERRUPTIBLE：等待资源有效，不可被其他进程中断**
    - ✓ **TASK_STOPPED：被调试暂停，或收到SIGSTOP等信号**
  - 不可运行态
    - ✓ **TASK_ZOMBIE：已退出而暂时没有被父进程收回资源的“僵尸”进程**
- 说明
  - 调度器主要处理的是可运行和被挂起两种状态下的进程

Linux 2.4的进程状态转换图



进程优先级

- 基本思想
  - 根据进程的价值和其对处理器时间的需求来对进程分级
    - 优先级高的进程先运行，低的后运行
    - 相同优先级的进程按**轮转**或**FIFO**方式进行调度
- **Linux**实现
  - 基于动态优先级的调度算法
    - 设置基本优先级，调度程序根据需要加、减优先级
    - 较长时间未分配到CPU的进程，通常**提高优先级**
    - 已经在CPU上运行了较长时间的进程，**降低优先级**
  - 两组独立的优先级范围
    - nice（-20~19）：决定分配给进程的时间片大小
    - 实时优先级（0~99）：实时进程的优先级高于普通进程

Linux 2.4中的进程优先级

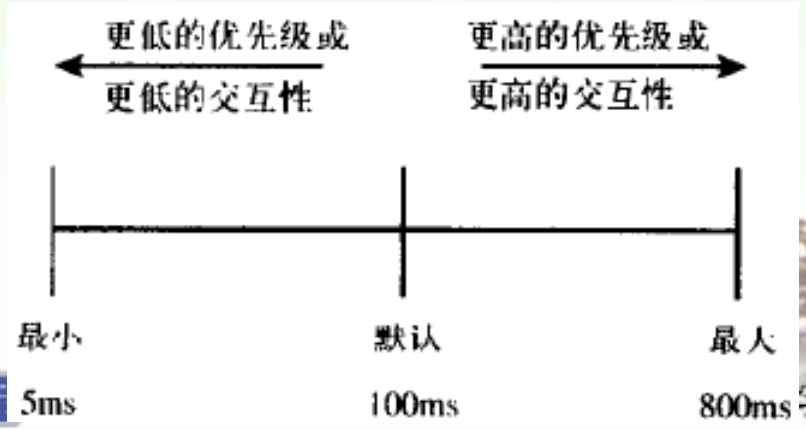
- 静态优先级：**priority（0~70）**
  - 代表分配给进程的时间片
  - 不随时间而改变，只能由用户进行修改，一般通过**nice**设定
  - 指明在被迫和其他进程竞争CPU之前，该进程所应该被允许的时间片的最大值。
- 动态优先级：**counter**
  - 只要进程拥有CPU，它就随着时间不断减小
  - 当它小于0时，标记进程重新调度
  - 指明在这个时间片中所剩余的时间量
- 实时优先级：**rt_priority（1~99）**
  - 指明这个进程自动把CPU交给哪一个其他进程，较高权值的进程总是优先于较低权值的进程
  - 如果一个进程不是实时进程，其优先级就是0，所以实时进程总是优先于非实时进程的

Linux 2.4中的优先级策略

- 普通进程
  - 基本思想：动态优先级调度
    - ✓ 周期性地修改进程的优先级（避免饥饿）
    - ✓ 根据进程的counter值
  - 基本过程
    - ✓ counter变为0的时，用priority对counter重新赋值
    - ✓ 但只有所有处于可运行状态的普通进程的时间片都用完以后，才对counter重新赋值
    - ✓ 普通进程运行过程中，counter的减小为其它进程提供运行机会，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
- 实时进程
  - 基本思想：静态优先级策略
    - ✓ counter只用来表示该进程的剩余时间片，不作为衡量其是否值得运行的标准（与普通进程的区别）

Linux调度策略

- 动态调整优先级及时间片长度的机制
  - 提高交互式程序的优先级，让它们运行得更频繁
  - 调度程序为交互式程序提供较长的默认时间片
  - 根据进程的优先级动态调整其时间片
  - 进程不一定要一次性使用完所有的时间片，可以分批使用，从而确保尽可能长时间地处于可运行状态
  - 没有时间片的进程不会再投入运行，除非等到其他所有进程都耗尽其时间片



Linux 2.4中的调度策略

- 域成员名
  - policy
- 策略分类
  - 实时进程
    - ✓SCHED_FIFO：先进先出调度，除非有更高优先级进程申请运行，否则该进程保持运行至退出才让出CPU
    - ✓SCHED_RR：轮转式调度，该进程被调度下来后将被放置于运行队列的末尾，以保证其他实施进程有机会运行
  - 普通进程
    - ✓SCHED_OTHER：常规的分时调度策略
  - 其他
    - ✓SCHED_YIELD：置位时表示主动放弃CPU

Linux 2.4进程调度的依据—权值

- 基本思想
  - 选择一个权值（weight）最大的进程
- 权值计算：goodness()
  - 综合policy，priority，rt_priority和counter四项计算
  - 普通进程的权值
    - ✓ weight = p->counter
  - 实时进程的权值
    - ✓ weight = 1000 + rt_priority

权值的计算—goodness()

- 位置
  - /kernel/sched.c
- 功能
  - 计算一个处于可运行状态的进程值得运行的程度
  - 运行队列中的每个进程每次执行schedule时都要调度它
- 返回值类型
  - <=1000：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
    - ✓ 在SMP情况下，SMP模式会优先照顾等待同一个处理器的进程
  - >1000：只能赋给“实时”进程
    - ✓ 不管是UP还是SMP，实时进程的goodness值的范围是从1001到1099。
- 说明
  - goodness() 不会返回负值
    - ✓ 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

影响权值的因素

- 普通进程
  - 进程当前时间片内所剩的tick数
    - 该既代表进程优先级，又反映进程的“欠运行程度”（weight = p->counter）
  - 进程上次运行的CPU是否就是当前CPU
    - 若是，权值增加一个常量（weight += PROC_CHANGE_PENALTY）
  - 切换是否需要切换内存
    - 若不需要，则权值加1(weight += 1)
  - 用户可见的优先级nice
    - nice越小则权值越大(weight += 20 - p->nice)。
- 实时进程
  - 权值大小仅由rt_priority值决定
    - weight = 1000 + p->rt_priority
    - 1000的基准量使得实时进程的权值比所有非实时进程都要大
    - 因此只要就绪队列中存在实时进程，调度器都将优先满足它的运行需要

goodness()函数分析

```
static inline int goodness(struct task_struct * p, int this_cpu, struct
mm_struct *this_mm) {
    int weight;
    weight = -1;
    if (p->policy & SCHED_YIELD)
        goto out;
    if (p->policy == SCHED_OTHER) {/*普通进程*/
        weight = p->counter;
        if (!weight) /* 进程的时间片已经用完，则直接转到标号out*/
            goto out;

#ifdef CONFIG_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
```

goodness()函数分析

```
/*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*/
        if (p->mm == this_mm || !p->mm)
            weight += 1;
        weight += 20 - p->nice;
        goto out;
    }
    weight = 1000 + p->rt_priority; /*实时进程*/
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*/
}
```

调度时期

- linux调度算法把CPU时间划分为时期（epoch）
  - 在一个单独的时期内，每个进程有一个指定的时间片
    - ✓一个进程用完它的时间片时，就会被强占
    - ✓只要进程的时间片没有用完，就可以被多次调度运行
  - 所有进程用完其时间片，一个时期才结束
  - 此时重新计算所有进程的时间片，并开始一个新的时期
- Linux的时间单位
  - 时钟滴答（10ms）
  - 时间片即指时间滴答数
  - 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。

时间片的选择

- 时间片的长短对系统性能非常关键，时间片大小的选择总是一种折衷，既不能太长也不能太短
  - 太短
    - ✓ 频繁切换会造成系统开销过大
    - ✓ 假如切换时间为1ms，时间片设置为1ms，则没空执行进程
  - 太长
    - ✓ 几乎每个进程都一次运行完，并发的概念基本消失
    - ✓ 导致系统对交互的响应表现欠佳
    - ✓ 普通进程需要等待很长时间才能运行
- Linux采取单凭经验的方法，即选择尽可能长的时间片，同时能保持良好的响应时间

时间片配额

- 为非SCHED_FIFO调度策略的每个进程提供一个运行的时间配额counter
- counter
  - 剩余的时间片
  - 每个进程的counter初值与nice值有关
  - 每次时钟中断(tick)发生，时间片都会减1，直到为0（则请求调度）。新时期开始时，将重新计算。
  - 创建一个新的进程时，子进程会继承父进程的一半剩余时间片。

```
p->counter = (current->counter + 1) >> 1;
current->counter >>= 1;
```

基本时间片的计算

- 基本时间片参数：**nice**
  - nice越小则counter越大，即优先级越高的进程所允许获得的CPU时间也相对越多。

nice缺省为0（在-20到19之间选择）

```
p->counter = (p->counter >> 1) + NICE_TO_TICKS(p->nice);
```

```
#define NICE_TO_TICKS(nice)  (TICK_SCALE(20-(nice))+1)
```

```
#define TICK_SCALE(x)     ((x) >> 2)
```

通常，基本时间片的值为6，由于时钟中断大约10ms左右，因此基本时间片的长度大约60ms

- NICE_TO_TICKS将系进程的优先级（nice）换算成时间配额
- 可以通过nice、setpriority系统调用调整进程的基本时间片

do fork()中与调度相关信息的设置

- 系统中除init_task是手工创建的以外，其他进程，包括其他CPU上的idle进程都是通过do_fork()创建的，所不同的是，创建idle进程时使用了CLONE_PID标志位。
- 在do_fork()中，新进程的属性设置为：
  - state：TASK_UNINTERRUPTIBLE
  - pid：如果设置了CLONE_PID则与父进程相同（仅可能为0），否则为下一个合理的pid
  - cpus_runnable：全1；未在任何cpu上运行
  - processor：与父进程的processor相同；子进程在哪里创建就优先在哪里运行
  - counter：父进程counter值加1的一半；同时父进程自己的counter也减半，保证进程不能通过多次fork来偷取更多的运行时间（同样，在子进程结束运行时，它的剩余时间片也将归还给父进程，以免父进程因创建子进程而遭受时间片的损失）

couter值的更新

- 在每次时钟中断时都要递减当前进程的counter值
- 更新函数：update_process_times()
  - 对于调度策略为SCHED_RR的进程，一旦counter降到0，就要从runqueue中当前的位置移到队列的末尾，同时恢复其最初的时间配额
  - 对于调度策略为SCHED_OTHER的进程，当所有这些进程的counter值降为0时，重新计算并设置每个进程的时间配额

update process times()函数代码分析

```
void update_process_times(int user_tick)
{
    struct task_struct *p = current;
    int cpu = smp_processor_id(), system = user_tick ^ 1;

    update_one_process(p, user_tick, system, cpu);
    if (p->pid) {
        if (--p->counter <= 0) {
            p->counter = 0;
            /*
             * SCHED_FIFO is priority preemption, so this is
             * not the place to decide whether to reschedule a
             * SCHED_FIFO task or not - Bhavesh Davda
             */
            if (p->policy != SCHED_FIFO) {
                p->need_resched = 1;
            }
        }
        if (p->nice > 0)
            kstat.per_cpu_nice[cpu] += user_tick;
        else
            kstat.per_cpu_user[cpu] += user_tick;
        kstat.per_cpu_system[cpu] += system;
    } else if (local_bh_count(cpu) || local_irq_count(cpu) > 1)
        kstat.per_cpu_system[cpu] += system;
} ? end update_process_times ?
```

非实时进程的couter值更新代码

```
repeat_schedule:
    /*
     * Default process to select..
     */
    next = idle_task(this_cpu);
    c = -1000;
    list_for_each(tmp, &runqueue_head) {
        p = list_entry(tmp, struct task_struct, run_list);
        if (can_schedule(p, this_cpu)) {
            int weight = goodness(p, this_cpu, prev->active_mm);
            if (weight > c)
                c = weight, next = p;
        }
    }

    /* Do we need to re-calculate counters? */
    if (unlikely(!c)) {
        struct task_struct *p;

        spin_unlock_irq(&runqueue_lock);
        read_lock(&tasklist_lock);
        for_each_task(p)
            p->counter = (p->counter >> 1) + NICE_TO_TICKS(p->nice);
        read_unlock(&tasklist_lock);
        spin_lock_irq(&runqueue_lock);
        goto repeat_schedule;
    }
```

调度器schedule()的主要函数和宏

- **schedule()**
  - 进程调度的主函数
- **switch_to()**
  - schedule()中调用，进行上下文切换的宏
- **reschedule_idle()**
  - 在SMP系统中，如果被切换下来的进程仍然是可运行的，则调用reschedule_idle()重新调度，以选择一个空闲的或运行着低优先级进程的CPU来运行这个进程
- **goodness()**
  - 优先级计算函数，选择一个最合适的进程投入运行

调度器schedule()的主要工作

- 从移走进程processor域读取cpu标识，存入局部变量this_cpu
- 初始化sched_data变量，指向当前处理器schedule_data结构
- 调用goodness()函数选取进程,对于上一次也在当前处理器的进程加上PROC_CHANGE_PENALTY的优先权
- 如果必要,重新计算动态优先权
- 设置sched_data->last_schedule值为当前时间
- 调用switch_to()宏执行切换
- 调用schedule_tail()，如果传入的prev进程仍是可运行的而且不是空闲进程，schedule_tail调用reschedule_idle()来选择一个合适的处理器

schedule()函数代码分析

```
asmlinkage void schedule(void)
{
    struct schedule_data * sched_data;
    struct task_struct *prev, *next, *p;
    struct list_head *tmp;
    int this_cpu, c;


    spin_lock_prefetch(&runqueue_lock);

    BUG_ON(!current->active_mm);
need_resched_back:
    prev = current;
    this_cpu = prev->processor;

    if (unlikely(in_interrupt())) {
        printk("Scheduling in interrupt\n");
        BUG();
    }

    release_kernel_lock(prev, this_cpu);
```

schedule()函数代码分析

```
/*
 * 'sched_data' is protected by the fact that we can run
 * only one process per CPU.
 */
sched_data = & aligned_data[this_cpu].schedule_data;

spin_lock_irq(&runqueue_lock);

/* move an exhausted RR process to be last.. */
if (unlikely(prev->policy == SCHED_RR))
    if (!prev->counter) {
        prev->counter = NICE_TO_TICKS(prev->nice);
        move_last_runqueue(prev);
    }

switch (prev->state) {
    case TASK_INTERRUPTIBLE:
        if (signal_pending(prev)) {
            prev->state = TASK_RUNNING;
            break;
        }
    default:
        del_from_runqueue(prev);
    case TASK_RUNNING:;
}
prev->need_resched = 0;
```

schedule()函数代码分析

```
repeat_schedule:
    /*
     * Default process to select..
     */
    next = idle_task(this_cpu);
    c = -1000;
    list_for_each(tmp, &runqueue_head) {
        p = list_entry(tmp, struct task_struct, run_list);
        if (can_schedule(p, this_cpu)) {
            int weight = goodness(p, this_cpu, prev->active_mm);
            if (weight > c)
                c = weight, next = p;
        }
    }

    /* Do we need to re-calculate counters? */
    if (unlikely(!c)) {
        struct task_struct *p;

        spin_unlock_irq(&runqueue_lock);
        read_lock(&tasklist_lock);
        for_each_task(p)
            p->counter = (p->counter >> 1) + NICE_TO_TICKS(p->nice);
        read_unlock(&tasklist_lock);
        spin_lock_irq(&runqueue_lock);
        goto repeat_schedule;
    }
```

schedule()函数代码分析

```
/*
 * from this point on nothing can prevent us from
 * switching to the next task, save this fact in
 * sched_data.
 */
sched_data->curr = next;
task_set_cpu(next, this_cpu);
spin_unlock_irq(&runqueue_lock);

if (unlikely(prev == next)) {
        /* We won't go through the normal tail, so do this by hand */
        prev->policy &= ~SCHED_YIELD;
        goto same_process;
}
```

schedule()函数代码分析

```
prepare_to_switch();
{
     struct mm_struct *mm = next->mm;
     struct mm_struct *oldmm = prev->active_mm;
     if (!mm) {
          BUG_ON(next->active_mm);
          next->active_mm = oldmm;
          atomic_inc(&oldmm->mm_count);
          enter_lazy_tlb(oldmm, next, this_cpu);
     } else {
          BUG_ON(next->active_mm != mm);
          switch_mm(oldmm, mm, next, this_cpu);
     }

     if (!prev->mm) {
          prev->active_mm = NULL;
          mmdrop(oldmm);
     }
}
```

schedule()函数代码分析

```
        /*
         * This just switches the register state and the
         * stack.
         */
        switch_to(prev, next, prev);
        __schedule_tail(prev);

same_process:
        reacquire_kernel_lock(current);
        if (current->need_resched)
                goto need_resched_back;
        return;
} ? end schedule ?
```

switch_to()的主要工作

- 将esi，edi，ebp压入堆栈
- 堆栈指针esp保存到prev->thread.esp
- 将esp恢复为next->thread.esp
- 将标号1：的地址保存到prev->thread.eip
- 将next->eip压入堆栈
- 无条件跳转到__switch_to()函数，切换LDT和fs，gs等寄存器，由于有了上一步的工作，因此本函数返回时已经切换到了next的上下文
- switch_to()中jmp指令以后的代码即标号1：的代码作的工作与第一步相反，即从堆栈弹出ebp，edi和esi。这部分的代码是下次该进程运行时最先执行的代码

switch_to() 的宏实现

```
#define switch_to(prev,next,last) do {\
    asm volatile("pushl %%esi\n\t"   \
    "pushl %%edi\n\t" \
    "pushl %%ebp\n\t"
    \esp保存到prev->thread.esp中
    "movl %%esp,%0\n\t"
    \从next->thread.esp恢复esp
    "movl %3,%%esp\n\t"
    \在prev->thread.eip中保存"1: "的跳转地址,
    \当prev被再次切换到的时候将从那里开始执行
    "movl $1f,%1\n\t"
    \在栈上保存next->thread.eip,
    \   switch_to()返回时将转到那里执行,
    \即进入next进程的上下文
    "pushl %4\n\t"
```

switch_to() 的宏实现

```
    \跳转到__switch_to(), 进一步处理(见下)
    "jmp __switch_to\n"
    "1:\t"                  \
    "popl %%ebp\n\t"        \
    "popl %%edi\n\t"        \
    \先恢复上次被切换走时保存
    \的寄存器值, 再从switch_to()中返回。
    "popl %%esi\n\t"
    :"=m" (prev->thread.esp), \%0
    "=m" (prev->thread.eip),\%1
    因为进程切换后, 恢复的栈上的prev信息
    不是刚被切换走的进程描述符, 因此此处
    使用ebx寄存器传递该值给prev
    "=b" (last) \ebx,
    :"m" (next->thread.esp),   \%3
    "m" (next->thread.eip), \%4
    "a" (prev), "d" (next), \eax,edx
    "b" (prev));   \ebx
} while (0)
```

schedule_tail()函数

- 执行时机
  - 完成切换后，调用该函数，但对于UP系统基本没什么影响
- 对于SMP系统
  - 被切换进程p仍然处于就绪态且未被任何CPU调度到，调用该函数为p挑选一个空闲的CPU，并强迫该CPU重新调度，以便将p重新投入运行
  - 进程从休眠状态中，调用该函数挑选一个合适的CPU运行，通过在wake_up_process()函数中调用reschedule_idle()实现的
  - 挑选CPU的原则如下
    - ✓p上次运行的CPU目前空闲
    - ✓所有空闲的CPU中最近最少活跃的一个
    - ✓CPU不空闲，但所运行的进程优先级比p的优先级低，且差值最大

schedule_tail()函数分析

```
static inline void __schedule_tail(struct task_struct *prev)
{
#ifdef CONFIG_SMP
    int policy;

    policy = prev->policy;
    prev->policy = policy & ~SCHED_YIELD;
    wmb();

    task_lock(prev);
    task_release_cpu(prev);
    mb();
    if (prev->state == TASK_RUNNING)
        goto needs_resched;

out_unlock:
    task_unlock(prev);
    return;
```

schedule_tail()函数分析

```
needs_resched:
    {
        unsigned long flags;
        if ((prev == idle_task(smp_processor_id())) ||
                        (policy & SCHED_YIELD))
            goto out_unlock;

        spin_lock_irqsave(&runqueue_lock, flags);
        if ((prev->state == TASK_RUNNING) && !task_has_cpu(prev))
            reschedule_idle(prev);
        spin_unlock_irqrestore(&runqueue_lock, flags);
        goto out_unlock;
    }
#else
    prev->policy &= ~SCHED_YIELD;
#endif /* CONFIG_SMP */
} ? end __schedule_tail ?
```

reschedule_idle()

- 当进程p变为可运行时，reschedule_idle()函数被调用
- 在SMP 系统上，这个函数决定进程是否应该抢占某一CPU上的当前进程。它执行下列操作
  - 先检查p进程上一次运行的cpu是否空闲，如果空闲，这是最好的cpu，直接返回
  - 找一个合适的cpu，查看SMP中的每个CPU上运行的进程，与p进程相比的抢先权，把具有最高的抢先权值的进程记录在target_task中，该进程运行的cpu为最合适的CPU
  - 如target_task为空，说明没有找到合适的cpu，直接返回。
  - 如果target_task不为空，则说明找到了合适的cpu，因此将target_task->need_resched置为1，如果运行target_task的cpu不是当前运行的cpu，则向运行target_task的cpu发送一个IPI中断，让它重新调度

reschedule_idle()函数代码分析

```
static void fastcall reschedule_idle(struct task_struct * p)
{
#ifdef CONFIG_SMP
    int this_cpu = smp_processor_id();
    struct task_struct *tsk, *target_tsk;
    int cpu, best_cpu, i, max_prio;
    cycles_t oldest_idle;

    /*
     * shortcut if the woken up task's last CPU is
     * idle now.
     */
    best_cpu = p->processor;
    if (can_schedule(p, best_cpu)) {
        tsk = idle_task(best_cpu);
        if (cpu_curr(best_cpu) == tsk) {
            int need_resched;
send_now_idle:
            /*
             * If need_resched == -1 then we can skip sending
             * the IPI altogether, tsk->need_resched is
             * actively watched by the idle thread.
             */
            need_resched = tsk->need_resched;
            tsk->need_resched = 1;
            if ((best_cpu != this_cpu) && !need_resched)
                smp_send_reschedule(best_cpu);
            return;
        }
    }
```

reschedule_idle()函数代码分析

```
/*
 * We know that the preferred CPU has a cache-affine current
 * process, lets try to find a new idle CPU for the woken-up
 * process. Select the least recently active idle CPU. (that
 * one will have the least active cache context.) Also find
 * the executing process which has the least priority.
 */
oldest_idle = (cycles_t) -1;
target_tsk = NULL;
max_prio = 0;

for (i = 0; i < smp_num_cpus; i++) {
    cpu = cpu_logical_map(i);
    if (!can_schedule(p, cpu))
        continue;
    tsk = cpu_curr(cpu);
```

reschedule_idle()函数代码分析

```
        /*
         * We use the first available idle CPU. This creates
         * a priority list between idle CPUs, but this is not
         * a problem.
         */
        if (tsk == idle_task(cpu)) {
#if defined(__i386__) && defined(CONFIG_SMP)
                        /*
             * Check if two siblings are idle in the same
             * physical package. Use them if found.
             */
            if (smp_num_siblings == 2) {
                if (cpu_curr(cpu_sibling_map[cpu]) ==
                        idle_task(cpu_sibling_map[cpu])) {
                    oldest_idle = last_schedule(cpu);
                    target_tsk = tsk;
                    break;
                }

                        }
#endif
```

reschedule_idle()函数代码分析

```
            if (last_schedule(cpu) < oldest_idle) {
                oldest_idle = last_schedule(cpu);
                target_tsk = tsk;
            }
        } ? end if tsk==idle_task(cpu) ?  else {
            if (oldest_idle == (cycles_t)-1) {
                int prio = preemption_goodness(tsk, p, cpu);

                if (prio > max_prio) {
                    max_prio = prio;
                    target_tsk = tsk;
                }
            }
        }
    } ? end for i=0;i<smp_num_cpus;i++ ?
```

reschedule_idle()函数代码分析

```
        tsk = target_tsk;
        if (tsk) {
            if (oldest_idle != (cycles_t)-1) {
                best_cpu = tsk->processor;
                goto send_now_idle;
            }
            tsk->need_resched = 1;
            if (tsk->processor != this_cpu)
                smp_send_reschedule(tsk->processor);
        }
        return;


#else /* UP */
        int this_cpu = smp_processor_id();
        struct task_struct *tsk;

        tsk = cpu_curr(this_cpu);
        if (preemption_goodness(tsk, p, this_cpu) > 0)
            tsk->need_resched = 1;
#endif
} ? end reschedule_idle ?
```

Linux2.4调度算法的缺点

- 每次调度时，调度器都要线性遍历这个队列
  - 系统中调度算法属于**O(n)**，开销是线性增长的
- 当大多数的就绪进程的时间片都用完，而又还投有重新分配时间片的时候，SMP系统中有些处理器处于空闲状态
  - 只有一个全局的就绪进程队列，对多处理器的伸缩性支持不好
  - 时间片的重算循环制约了多处理器的效率
- 空闲处理器执行时间片尚未用尽，而处于等待状态的进程时，会导致进程开始在处理器之间“跳跃”
  - 处理器的亲和性不好，实时进程或者占用内存大的进程在处理器之间跳跃会严重影响系统的性能

主要内容

- 背景知识
  - 调度策略和调度机制
  - Linux 2.4的进程调度机制
  - Linux 2.6的进程调度机制
- 实验内容
  - 将Linux 2.6调度算法修改成随机调度算法

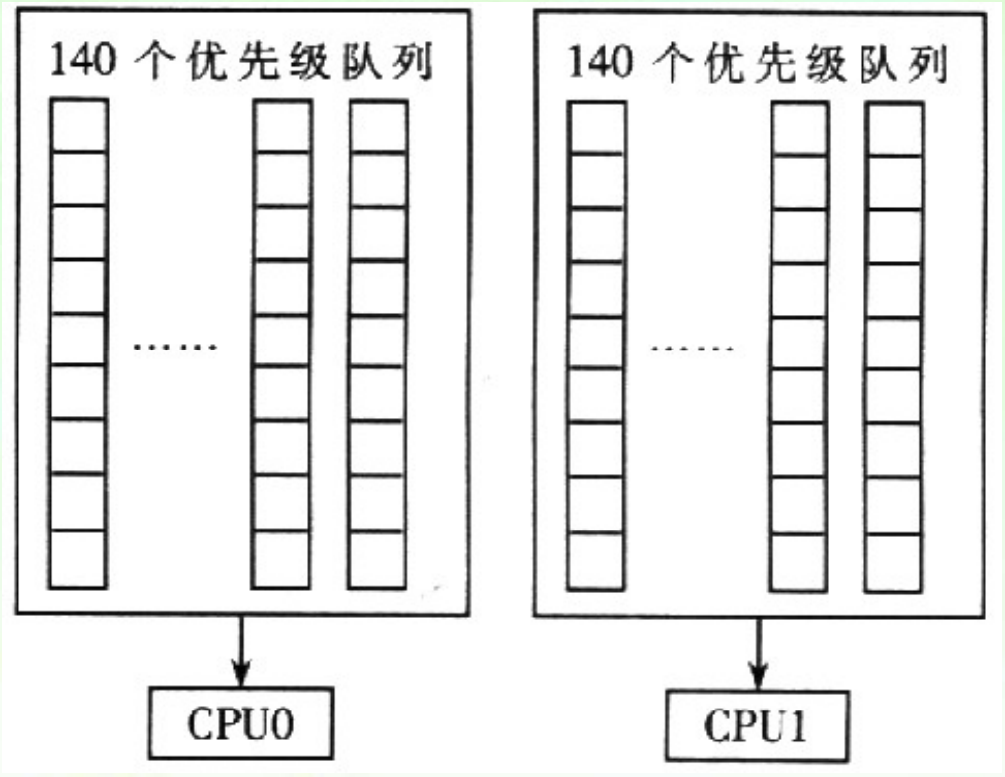
Linux 2.6的进程调度

- 进程调度结构
- 调度相关的基本数据
- 调度策略
- 调度时机
- **schedule()分析**

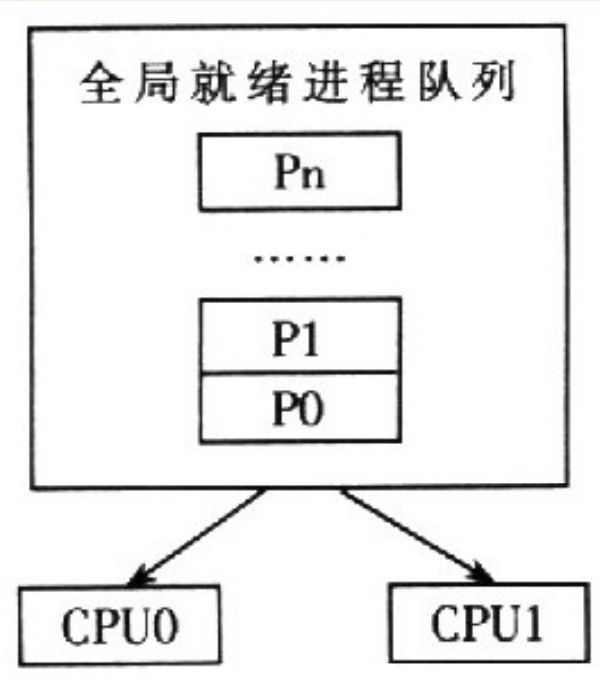
Linux 2.6调度算法优化目标

- 提供完全的O(1)调度算法，也就是说，不管系统中进程数量的多少，调度器中所有的算法都必须在常数时间内完成
- 应该对SMP有良好的可伸缩性，理想情况下，每个处理器应该有独立的可执行进程队列和锁机制
- 应该提高SMP的处理器亲和性，但是同时也应该有在负载不平衡的时候在处理器间迁移进程的能力

Linux 2.6的调度结构



Linux 2.6



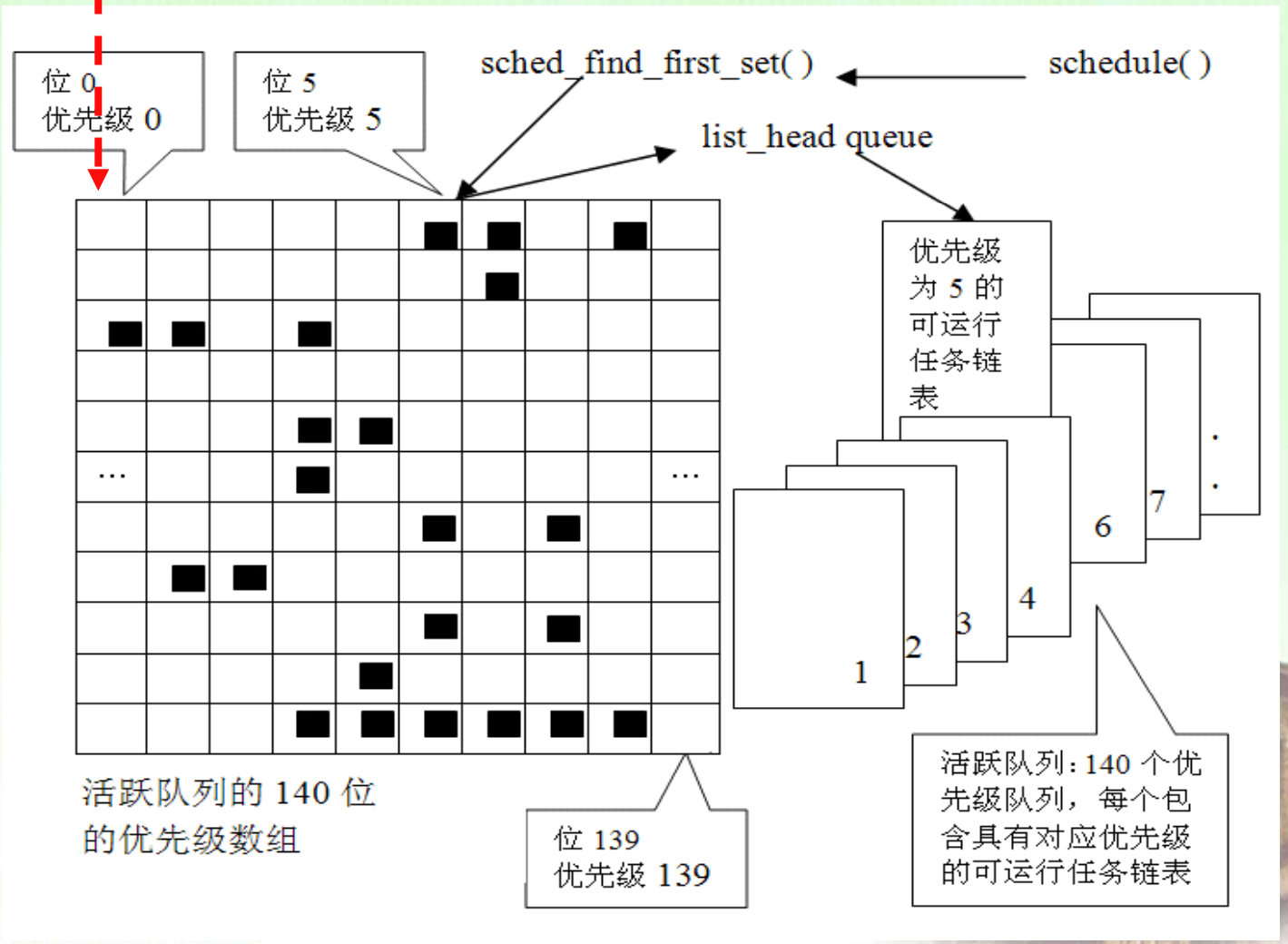
Linux 2.4

Linux2.6的调度策略设计

- 基于每个CPU来分布时间片，取消全局同步和重算循环
- 每个处理器有两个数组，活动就绪进程队列数组和不活跃就绪进程队列数组
  - 如果一个进程消耗完了它的“时间片”，就进入不活跃就绪进程数组的相应队列的队尾
  - 当所有的进程都“耗尽”了它的“时间片”后，交换活跃与不活跃就绪进程队列数组，不需要任何其他的开销
- 每个数组中有140个就绪进程队列(runqueue)，每个队列对应于140个优先级的某一个
  - 通过位图标记队列状态，调度时，通过find_first_bit找到第一个不为空的队列，并取队首的进程即可
  - 不管队列中有多少个就绪进程，挑选就绪程的速度是一定的，所以称为0(1)算法

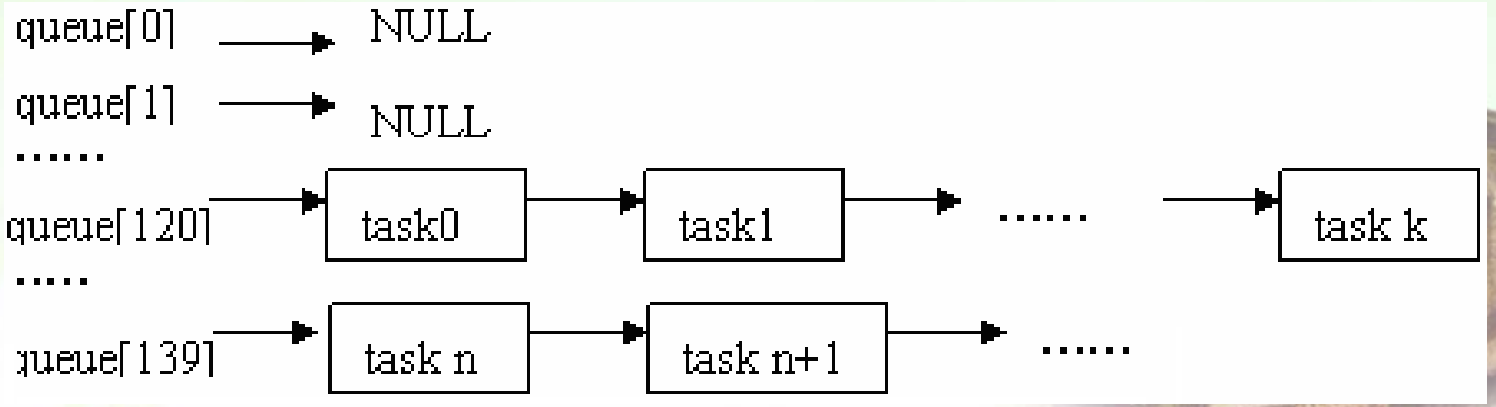
Linux 2.6 O(1)级调度算法结构

prio_array_t arrays[2];



Linux 2.6 优先级数组数据结构

- prio_array_t *active, *expired, arrays[2]
  - 根据时间片是否被用完将就绪队列分成两类
    - active：时间片没有用完，当前可被调度的就绪进程
    - expired：时间片已用完的就绪进程
  - arrays 二元数组是两类就绪队列的容器，active 和 expired 分别指向其中一个
    - active 中的进程一旦用完了自己的时间片，就被转移到 expired 中，并设置好新的初始时间片
    - 而当 active 为空时，则表示当前所有进程的时间片都消耗完了，此时，active 和 expired 进行一次对调，重新开始下一轮的时间片递减过程



Linux 2.6就绪队列状态的交换

- 交换时机
  - 活动数组内的可执行队列上的进程为空
- 时间片计算时机
  - 进程由活动数组移至过期数组之前
- 交换方法（在schedule()中）

```
struct prio_array * array = rq->active;
if (!array->nr_active) {
        rq->active = rq->expired;
        rq->expired = array;
}
```

Linux 2.6优先级数组定义

❖struct prio_array

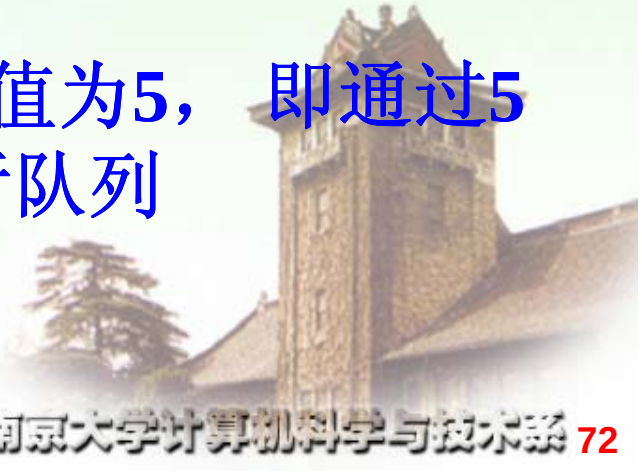
```
typedef struct runqueue runqueue_t;

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 映 射 */
    struct list_head queue[MAX_PRIO];/*以 优 先 级 为 索 引 的 HASH表 */
};
```

- queue是指定优先级进程list的指针，如queue[i]就是priority为 i 的进程的指针
- bitmap是一张优先级的位图，每一位代表了一个优先级
- MAX_PRIO指的是优先级的数量

Linux 2.6 中MAX_PRIO的定义

```
#define BITMAP_SIZE ((((MAX_PRIO+1+7)/8)+sizeof(long)-1)/sizeof(long))
```

- 1+7的解释
  - 优先级0～MAX_PRIO（140）之间，共有MAX_PRIO+1个优先级，因此要加1
  - 被除数为8，加7是为了向上取整
- sizeof(long)-1的解释
  - 被除数为sizof(long)，先加上sizeof(long)-1，也是为了上取整
- 通过以上计算，BITMAP_SIZE的值为5，即通过5个四字节的整数位(160位)作为运行队列queue[MAX_PRIO]的位图掩码



sched find first bit函数

```
static inline int sched_find_first_bit(const unsigned long *b)
{
        if (unlikely(b[0]))
                return __ffs(b[0]);
        if (unlikely(b[1]))
                return __ffs(b[1]) + 32;
        if (unlikely(b[2]))
                return __ffs(b[2]) + 64;
        if (b[3])
                return __ffs(b[3]) + 96;
        return __ffs(b[4]) + 128;
}
```

- _ffs()用来查找一个长整型变量最右边的1是第几位

```
static inline unsigned long __ffs(unsigned long word)
{
        asm("bsf %1,%0"
                : "=r" (word)
                : "rm" (word));
        return word;
}
```

Linux 2.6 O(1)级调度算法结构小结

- 按宏定义，会生成160bit的位数组（5个long)，构成一张表
- 每一位对应一个优先级，该优先级有active状态（时间片未用完）的，在相应位置1, 否则置0
- 从右向左位扫描1，得到优先级最高的优先级号k
- 进入queue[k]数组

O(1) 调度算法执行过程

- 查找过程分解为 **n** 步，每一步所耗费的时间都是 **O(1)** 量级的
- **prio_array** 中包含一个就绪队列数组，数组的索引是进程的优先级（共 **140** 级），相同优先级的进程放置在相应数组元素的链表 **queue** 中
- 调度时直接给出就绪队列 **active** 中具有最高优先级的链表中的第一项作为候选进程
- 优先级的计算过程则分布到各个进程的执行过程中进行
- 位映射数组用于加速寻找存在就绪进程的链表，其中每位对应一个优先级链表
  - 如果该优先级链表非空，则对应位为 **1**，否则为 **0**
  - 核心还要求每个体系结构都构造一个 **sched_find_first_bit()** 函数来执行这一搜索操作，快速定位第一个非空的就绪进程链表

O(1) 调度算法执行的核心代码

```
struct task_struct *prev, *next;
struct list_head *queue;
struct prio_array *array;
int idx;

prev = 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);
```

Linux 2.6调度器的优点

- 每个处理器都有独立的就绪进程队列，各个处理器可以并行运行调度程序来挑选进程运行，不同处理器上的进程可以完全并行地休眠、唤醒和上下文切换
- 进程只映射到一个处理器的就绪进程队列中，不会被其他的处理器选中，因而也就不会在不同的处理器之间跳跃

Linux 2.6运行队列

❖定义位置：**kernel/sched.c**

```
struct runqueue {
spinlock_t          lock;                /* 保护运行队列的自旋锁*/
unsigned long       nr_running;          /* 可运行任务数目 */
unsigned long       nr_switches;         /* 上下文切换数目*/
unsigned long       expired_timestamp;   /* 队列最后被换出时间 */
unsigned long       nr_uninterruptible;  /* 处于不可中断睡眠状态的任务数目*/
unsigned long long  timestamp_last_tick; /* 最后一个调度程序的节拍 */
struct task_struct  *curr;               /* 当前运行任务 */
struct task_struct  *idle;               /* 该处理器的空任务*/
struct mm_struct    *prev_mm;            /* 最后运行任务的mm_struct结构体*/
struct prio_array   *active;             /* 活动优先级队列 */
struct prio_array   *expired;            /* 超时优先级队列 */
struct prio_array   arrays[2];           /* 实际优先级数组*/
struct task_struct  *migration_thread;   /* 移出线程 */
struct list_head    *migration_queue;    /* 移出队列*/
atomic_t            nr_iowait;           /* 等待I/O操作的任务数目 */
};
```

Linux 2.6运行队列结构说明

- **task_t *curr**
  - 本 CPU 正在运行的进程
- **tast_t *idle**
  - 指向本 CPU 的 idle 进程，相当于 2.4 中 **init_tasks[this_cpu()]**
- **struct mm_struct *prev_mm**
  - 保存进程切换后被调度下来的进程（称之为 prev）的 active_mm 结构指针
    - ✓ *在 2.6 中 prev 的 active_mm 在进程切换完成后释放*
- **unsigned long nr_running**
  - 本 CPU 上的就绪进程数，等于active 和 expired 两个队列中进程数的总和，是说明本 CPU 负载情况的重要参数
- **unsigned long nr_switches**
  - 记录本 CPU 上自调度器运行以来发生的进程切换的次数
- **unsigned long nr_uninterruptible**
  - 本 CPU处于 TASK_UNINTERRUPTIBLE 状态的进程数，和负载信息有关

Linux 2.6运行队列结构说明

- **int best_expired_prio**
  - 记录 expired 就绪进程组中的最高优先级，该变量在进程进入 expired 队列的时候保存（schedule_tick()
- **unsigned long expired_timestamp**
  - 记录最早发生的进程耗完时间片事件的时间，用来表征 expired 中就绪进程的最长等待时间
  - 一般情况下，时间片结束的进程应该从 active 队列转移到 expired 队列中，但对交互式进程，调度器就会让其保持在 active 队列上以提高它的响应速度
  - 但如果 expired 队列中的进程已经等待足够长时间，即使是交互式进程也应该转移到 expired 队列上来，排空 active
  - 这个阀值就体现在EXPIRED_STARVING(rq) 上：如果以下两个条件都满足，则 EXPIRED_STARVING() 返回真
    - ✓ expired 队列中至少有一个进程已经等待了足够长的时间：
      （当前绝对时间 - expired_timestamp） >=
      （STARVATION_LIMIT * 队列中所有就绪进程总数 + 1）；
    - ✓ 运行进程的静态优先级比 expired 队列中最高优先级要低（best_expired_prio，数值要大）

Linux 2.6运行队列结构说明

- **atomic_t nr_iowait**
  - 本 CPU 因等待 IO 而处于休眠状态的进程数
- **unsigned long timestamp_last_tick**
  - 本就绪队列最近一次发生调度事件的时间，在负载平衡的时候会用到
- **int prev_cpu_load[NR_CPUS]**
  - 进行负载平衡时各个 CPU 上的负载状态（此时就绪队列中的 nr_running 值），以便分析负载情况
- **atomic_t *node_nr_running; int prev_node_load[MAX_NUMNODES]**
  - 各个 NUMA 节点上就绪进程数和上一次负载平衡操作时的负载情况
- **task_t *migration_thread**
  - 指向本 CPU 的迁移进程。每个 CPU 都有一个核心线程用于执行进程迁移操作
- **struct list_head migration_queue**
  - 需要进行迁移的进程列表

Linux 2.6可执行运行队列的管理

- 获取可执行队列
  - cpu_rq(processor)：获取给定处理器可执行队列指针
  - this_rq()：返回当前处理器的可执行队列
  - task_rq(task)：返回给定任务所在的队列指针
- 运行队列的自旋锁机制
  - 成员名：lock
  - 当需要对 runqueue 进行操作时，仍然应该锁定，但这个锁定操作只影响一个 CPU 上的就绪队列。
  - 加锁
    - ✓ task_rq_lock()
    - ✓ this_rq_lock()：锁住当前可执行队列
  - 解锁
    - ✓ task_rq_unlock()
    - ✓ rq_unlock(struct runqueue *rq)：释放给定队列上的锁

Linux 2.6运行队列的死锁避免机制

- 锁住多个运行队列的代码必须按同样的顺序获取这些锁：如按可执行队列地址从低到高顺序

```
/* 锁定 ... */
if (rq1 == rq2)
        spinlock(&rq1->lock);
else{
        if (rq1 < rq2){
                spin_lock(&rq1->lock);
                spin_lock(&rq2->lock);
        } else {
                spin_lock(&rq2->lock);
                spin_lock(&rq1->lock);
        }
}
```

```
/* 操作两个运行队列 ... */

/* 释放锁... */
spin_unlock(&rq1->lock);
if (rq1 != rq2)
        spin_unlock(&rq2->lock);
```

Linux 2.6进程描述符中与调度相关的成员

变量名	说明
state	进程的当前状态
cpus_allowed	以位向量形式表示可执行该进程的CPU
policy	进程的调度类型
rt_priority	进程的实时优先级
prio	进程的动态优先级
static_prio	进程的静态优先级
sleep_avg	进程的平均睡眠时间
time_slice	当前时间片中剩余的时钟节拍数，类似于2.4中的count
first_time_slice	如果进程肯定不会用完其时间片，则将该标志设为1
interactive_credit	记录进程的交互程度

Linux 2.6进程描述符中与调度相关的成员

变量名	说明
activated	进程被唤醒时所使用的条件码
timestamp	进程最近插入运行队列的时间或涉及本进程的最近一次进程切换时间
run_list	指向进程所属的运行队列中的下一个和前一个元素
array	记录当前CPU的活跃就绪队列（runqueue::active）
thread_info	当前进程的一些运行环境信息

Linux 2.6中的进程状态

- 域成员名：state

```
00105: #define TASK_RUNNING          0
00106: #define TASK_INTERRUPTIBLE    1
00107: #define TASK_UNINTERRUPTIBLE  2
00108: #define TASK_STOPPED          4
00109: #define TASK_TRACED           8
00110: #define TASK_ZOMBIE           16
00111: #define TASK_DEAD             32
```

- 主要变化
  - 宏定义数值上有较大的差别
  - 新增两种状态：TRACED、DEAD
    - ✓TASK_DEAD是表示已经退出且不需父进程回收的进程的状态。
    - ✓TASK_TRACED供调试使用

Linux 2.6中的进程唤醒时机

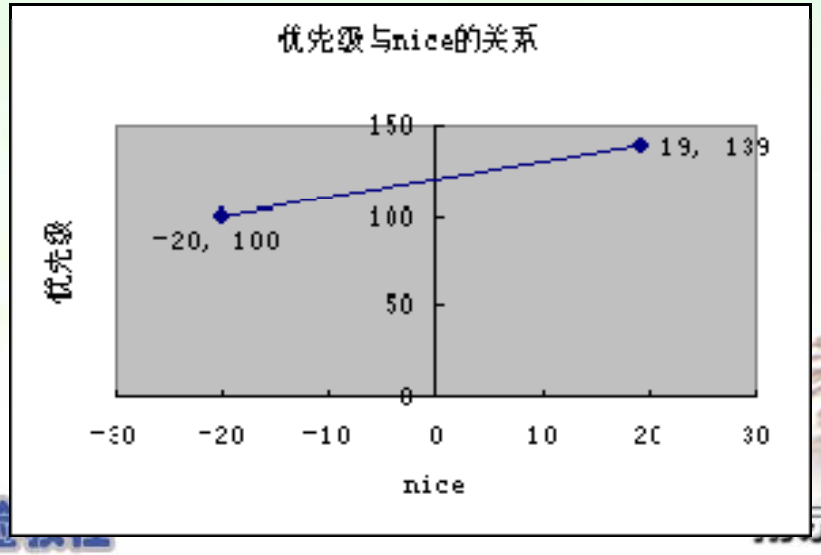
- 域成员名：activated
- 取值
  - -1，从 TASK_UNINTERRUPTIBLE 状态被唤醒
  - 0，缺省值，进程原本就处于就绪态
  - 1，从 TASK_INTERRUPTIBLE 状态被唤醒，且不在中断上下文中
  - 2，从 TASK_INTERRUPTIBLE 状态被唤醒，且在中断上下文中
- activated的修改时机
  - schedule() 中：被恢复为 0
  - activate_task()中：由 try_to_wake_up() 函数调用，用于激活休眠进程
    - ✓ 如果是中断服务程序调用的 activate_task()，即进程由中断激活，则该进程最有可能是交互式的，因此，置 activated=2；否则置 activated=1
    - ✓ 如果进程是从 TASK_UNINTERRUPTIBLE 状态中被唤醒的，则 activated=-1

Linux 2.6中的进程调度发生时间

- 域成员名：**timestamp**
- 调度事件发生时间分类
  - 被唤醒的时间（在 activate_task() 中设置）
  - 被切换下来的时间（在schedule()中设置）
  - 被切换上去的时间（在schedule()中设置）
  - 负载平衡相关的赋值
- 作用
  - 该值与当前时间的差值中可以分别获得“在就绪队列中等待运行的时长”、"运行时长"等与优先级计算相关的信息

Linux 2.6进程的静态优先级

- 域成员名：static_prio
  - 与2.4的nice值意义相同，但转换到与 prio 相同的取值区间。
  - 静态优先级决定了进程的初始时间片大小，这一点对实时进程及非实时进程都一样。但实时进程的static_prio 不参与优先级计算。
  - nice 与 static_prio 之间的关系如下:
    - ✓ static_prio = MAX_RT_PRIO + nice + 20
  - nice 与 static_prio的转换函数
    - ✓ PRIO_TO_NICE()、NICE_TO_PRIO()



Linux 2.6进程的动态优先级

- 域成员名：prio
  - 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）
    - ✓0~MAX_RT_PRIO-1（MAX_RT_PRIO 定义为100）属于实时进程范围
    - ✓MAX_RT_PRIO~MX_PRIO-1 属于非实时进程
  - 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，再通过上面描述的 priority_array 结构自动排序

Linux 2.6进程的时间片相关参数

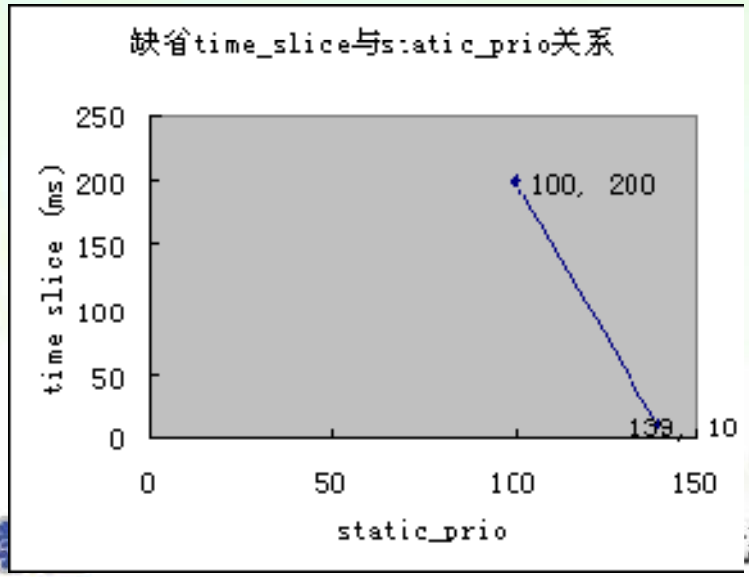
- 剩余时间片
  - 域成员名：time_slice
  - 相当于 2.4 的 counter，但不直接影响进程的动态优先级
- 是否首次拥有时间片
  - 域成员名：first_time_slice
  - 0 或 1，表示是否是第一次拥有时间片（刚创建的进程）
  - 该变量用来判断进程结束时是否应当将自己的剩余时间片返还给父进程

Linux 2.6进程的时间片基准值

- 进程的缺省时间片与进程的静态优先级相关

  MIN_TIMESLICE + ((MAX_TIMESLICE - MIN_TIMESLICE) * (MAX_PRIO-1 - (p)->static_prio) / (MAX_USER_PRIO-1))

  - **100~139 的优先级映射到 200ms~10ms 的时间片上去**
  - **AVG_TIMESLICE为2.6内核定义的平均时间片，相当于2.4中nice值为 0 时的时间片长度，此时根据上述公式计算所得大约是 102ms（2.4为60ms）**



Linux 2.6进程时间片变化

- 进程创建（do_fork()）
  - 与2.4 类似，子进程被创建时并不分配自己的时间片，而是与父进程平分父进程的剩余时间片
- 进程运行（sched_tick()）
  - time_slice值递减，一旦为 0，则按 static_prio 值重新赋予基准值，并请求调度
- 进程退出（sched_exit()）
  - 根据 first_time_slice 的值判断自己是否从未重新分配过时间片
    - ✓若是，则将自己的剩余时间片返还给父进程（保证不超过 MAX_TIMESLICE）。这个动作使进程不会因创建短期子进程而受到惩罚
    - ✓如果进程已经用完从父进程那分得的时间片，则不返还（2.4 未考虑）

Linux 2.6进程时间片对调度的影响

- 2.6 以时间片是否耗尽为标准将就绪进程分成 active、expired 两大类
  - 仅当active 进程时间片都耗尽，expired 进程才有机会运行
- 在 active 中挑选进程时，剩余时间片不作为影响调度优先级的一个因素
- 为满足内核可剥夺的要求，时间片太长的非实时交互式进程将被分成多个运行粒度运行
  - 每一段运行结束后，它都从 cpu 上被剥夺下来，放置到对应的 active 就绪队列的末尾，为其他具有同等优先级的进程提供运行的机会
  - 该操作在 schedule_tick() 对时间片递减之后进行，被强制从 cpu 剥夺，重新入队等候下一次调度的条件
    - ✓ 进程当前在 active 就绪队列中
    - ✓ 该进程是交互式进程
    - ✓ 该进程已经耗掉的时间片正好是运行粒度的整数倍
    - ✓ 剩余时间片不小于运行粒度

Linux 2.6进程运行粒度的定义

- 宏名称：**TIMESLICE_GRANULARITY**
  - 被定义为与进程的 sleep_avg 和系统总 CPU 数相关的宏。
  - sleep_avg 实际上代表着进程的非运行时间与运行时间的差值，与交互程度判断关系密切
  - 运行粒度说明内核的以下两个调度策略
    - ✓ 进程交互程度越高，运行粒度越小，这是交互式进程的运行特点所允许的；与之对应，CPU-bound 的进程为了避免 Cache 刷新，不应该分片
    - ✓ 系统 CPU 数越多，运行粒度越大

Linux 2.6优先级的计算过程

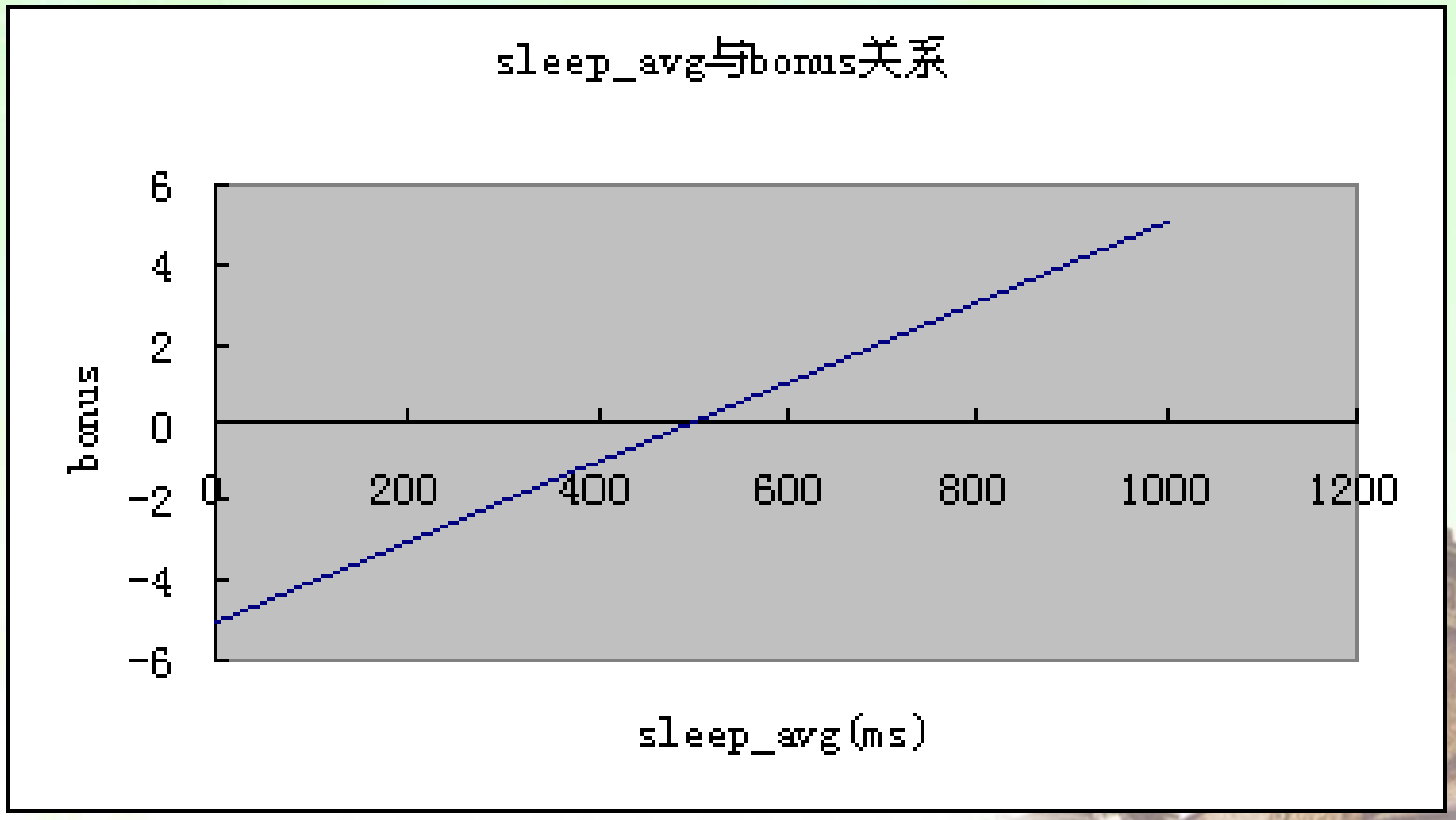
❖动态优先级计算

- 主要由 **effect_prio()** 函数完成（2.4由goodness()完成）
  - ✓ 非实时进程的优先级取决于静态优先级及进程的 sleep_avg
  - ✓ 实时进程的优先级实际上是在 setscheduler() 中设置的，且一经设定就不再改变
- 2.6 的动态优先级算法的实现关键在 sleep_avg 变量上
  - ✓ 在 effective_prio() 中，sleep_avg 的范围是 0~MAX_SLEEP_AVG
  - ✓ 该值将通过公式计算转换成bonus
- 动态优先级计算公式
  - ✓ 动态优先级= bonus - 静态优先级
  - ✓ 取值范围：MAX_RT_PRIO ~ MAX_PRIO 之间

Linux 2.6的bonus的计算

❖计算公式

(NS_TO_JIFFIES((p)->sleep_avg) * MAX_BONUS / MAX_SLEEP_AVG) - MAX_BONUS/2



Linux 2.6中sleep_avg与bonus关系

- 取值范围
  - -MAX_BONUS/2 ~ MAX_BONUS/2
- MAX_BONUS 定义
  - MAX_USER_PRIO*PRIO_BONUS_RATIO/100
- 说明
  - sleep_avg 对动态优先级的影响仅在静态优先级的用户优先级区（100~140）的1/4区间（±5）之内
  - 相对而言，静态优先级在优先级计算的比重要大得多
  - 这也是 2.6 调度系统中变化比较大的一个地方，调度器倾向于更多地由用户自行设计进程的执行优先级

Linux 2.6进程优先级的计算时机

- 进程状态改变
  - 创建进程
    - ✓ 子进程继承父进程的动态优先级，并添加到父进程所在的就绪队列中(wake_up_forked_process())
    - ✓ 若父进程不在任何就绪队列中，调用effect_prio() 函数计算优先级，并放置到相应就绪队列
  - 唤醒休眠进程
    - ✓ 调用 recalc_task_prio()设置从休眠状态中醒来的进程的动态优先级，再根据优先级放置到相应就绪队列中
  - 从 TASK_INTERRUPTIBLE 状态中被唤醒的进程
    - ✓ 已选定运行进程，调用 recalc_task_prio() 对该进程的优先级进行修正
  - 进程因时间片相关的原因被剥夺 cpu
    - ✓ 在 schedule_tick()中调用effect_prio() 重新计算优先级，重新入队
  - 其它时机
    - ✓ 包括 IDLE 进程初始化（init_idle()）、负载平衡（move_task_away()）以及修改 nice 值（set_user_nice()）、修改调度策略（setscheduler()）等主动要求改变优先级的情况

Linux 2.6中的进程平均等待时间

- 域成员名：**sleep_avg**
  - 进程等待时间与运行时间的差值
  - 初值为 0，在 0 到 NS_MAX_SLEEP_AVG 之间取值
- 作用
  - sleep_avg 反映了调度系统的两个策略：交互式进程优先和分时系统的公平共享
  - 既可用于评价该进程的“交互程度”，又可用于表示该进程需要运行的紧迫性
  - 该值是动态优先级计算的关键因子，sleep_avg 越大，计算出来的进程优先级也越高
    - ✓ 使得宏观上相同静态优先级的所有进程的等待时间和运行时间的比值趋向一致，反映了 Linux 要求各进程分时共享 cpu 的公平性

Linux 2.6内核修改sleep_avg的时机

- 内核对 sleep_avg 的修改
  - 从休眠状态唤醒时
    - ✓ activate_task()调用 recalc_task_prio()
  - 从TASK_INTERRUPTIBLE 状态唤醒后第一次调度到
    - ✓ schedule()中调用 recalc_task_prio()
  - 进程从 CPU 上被剥夺下来
    - ✓ schedule()
  - 进程创建
    - ✓ wake_up_forked_process()
  - 和进程退出
    - ✓ sched_exit()

Linux 2.6对交互式进程的考虑

- 内核有四处对交互式进程的优先考虑
  - sleep_avg
    - 交互式进程因为休眠次数多、时间长，sleep_avg 也会相应更大一些
  - interactive_credit
    - 表征该进程是否是交互式进程
  - TASK_INTERACTIVE()宏
  - 就绪等待时间的奖励

Linux 2.6的interactive credit

- 交互式进程判断条件
  - interactive_credi t> CREDIT_LIMIT
- interactive_credit 的递增（recalc_task_prio() 函数中）
  - 用户进程，不是从 TASK_UNINTERRUPTIBLE 被唤醒，且 sleep_time > INTERACTIVE_SLEEP(p)
  - 除以上情况外，sleep_avg 经过 sleep_time 调整后，如果大于 NS_MAX_SLEEP_AVG。
  - 一旦 interactive_credit 超过CREDIT_LIMIT，将不再增加
- interactive_credit 的递减（schedule() 函数）
  - 被切换进程的sleep_avg经运行时间修正后，若满足
    - ✓ sleep_avg <= 0
    - ✓ -CREDIT_LIMIT <= interactive_credit <= CREDIT_LIMIT
- 说明
  - 只有进程多次休眠，且休眠的时间足够长（长于运行的时间，长于“交互式休眠”时间），进程才有可能被列为交互式进程
  - 一旦被认为是交互式进程，则永远按交互式进程对待

Linux 2.6对交互式进程的优先级奖励

- 当进程从 **cpu** 上调度下来的时侯，如果是交互式进程，则它参与优先级计算的运行时间会比实际运行时间小，以此获得较高的优先级
- 交互式进程处于 **TASK_UNINTERRUPTIBLE** 状态下的休眠时间也会叠加到 **sleep_avg** 上，从而获得优先级奖励

Linux 2.6的TASK_INTERACTIVE()

- 判断条件
  - TASK_INTERACTIVE() prio <= static_prio-DELTA(p)
- 转换后形式
  - nice/4+2 <= bonus
    - ✓ 其中-20 <= nice <= 19, -5<=bonus<=+5
    - ✓ nice 大于 12 时，此不等式恒假，此时进程永远不会被当作交互式进程看待
- 作用
  - 当进程时间片耗尽时，如果该宏返回真，则该进程可能不进入 expired 队列而是保留在 active 队列中，以便尽快调度到这一交互式进程

Linux 2.6对交互式进程的调度处理

```
struct task_struct *task;
struct runqueue *rq;

task = current;
rq = this_rq();

if (!--task->time_slice) {
  if (!TASK_INTERACTIVE(task) || EXPIRED_STARVING(rq))
    enqueue_task(task, rq->expired);
  else
    enqueue_task(task, rq->active);
}
```

Linux 2.6的调度流程

- 清理当前运行中的进程（prev）
  - 更新sleep_avg：减去运行时间
  - 更新timestamp：当前时间，即记录被切换时间
  - 说明
    - ✓ 虽sleep_avg被修改，但在就绪队列位置不会改变，直至发生状态改变
- 选择下一个投入运行的进程（next）
  - active 就绪队列中优先级最高且等待时间最久的进程
  - 当前 runqueue 中没有就绪进程，则启动负载平衡从别的 cpu 上转移进程，再进行挑选;
  - 如果仍然没有就绪进程，则将本 cpu 的 IDLE 进程设为候选。
  - 对于next
    - ✓ 若从 TASK_INTERRUPTIBLE 休眠中醒来后第一次被调度到，则重新调整进程的优先级，并存入就绪队列中新的位置。
    - ✓ 除 sleep_avg 和 prio 的更新外，next 的 timestamp 也更新为当前时间，用于下一次被切换下来时计算运行时长。
- 设置新进程的运行环境
- 执行进程上下文切换
- 后期整理

Linux 2.6对内核抢占运行的支持

- **2.6 内核支持抢占运行，没有锁保护的任何代码段都有可能被中断**
  - 无论是返回用户态还是返回核心态，都有可能检查 **NEED_RESCHED** 的状态
  - 返回核心态时，只要 **preempt_count** 为 **0**，即当前进程目前允许抢占，就会根据 **NEED_RESCHED** 状态选择调用 **schedule()**
  - 在核心态中，时钟中断可不断发生，因此，只要有进程设置了当前进程的 **NEED_RESCHED** 标志，当前进程马上就有可能被抢占，而无论它是否愿意放弃 **cpu**，即使是核心进程也是如此。

Linux 2.6的负载平衡机制

❖Linux 2.4的负载平衡基本策略

- 基本方法
  - ✓进程p被切换下来之后，如果还有 cpu 空闲，或者该 cpu 上运行的进程优先级比自己低，那么 p 就会被调度到那个 cpu 上运行，核心正是用这种办法来实现负载的平衡
- 缺点
  - ✓进程迁移比较频繁，交互式进程（或高优先级的进程）可能还会在 cpu 之间不断“跳跃”。

❖Linux 2.6的基本策略

- 采用相对集中的负载平衡方案，分为“推”和“拉”两类操作

Linux 2.6负载均衡的“拉”操作

- 基本思想
  - 当某个 cpu 负载过轻而另一个 cpu 负载较重时，系统会从重载 cpu 上“拉”进程过来
  - “拉”的负载平衡操作实现在 load_balance() 函数中，有两种调用方式
    - ✓ “忙平衡”：当前 cpu 不空闲
    - ✓ “空闲平衡”：当前 cpu空闲
  - 拉"进程的具体动作在pull_task()函数中实现

Linux 2.6负载均衡的“忙平衡”

- 基本方法
  - 时钟中断周期性启动load_balance()
- Linux 2.6 倾向于尽可能不做负载平衡，“拉”操作的限制
  - 最繁忙 cpu 的负载超过当前 cpu 负载的 25% 时，才进行负载平衡;
  - 当前 cpu 负载取当前真实负载和上一次执行负载平衡时的负载的较大值，平滑负载凹值
  - 各 cpu 的负载情况取当前真实负载和上一次执行负载平衡时的负载的较小值，平滑负载峰值
  - 对源、目的两个就绪队列加锁之后，再确认一次源就绪队列负载没有减小，否则取消负载平衡动作
- 迁移方法
  - 找到最繁忙的 cpu（源 cpu）之后，确定需要迁移的进程数为源 cpu 负载与本 cpu 负载之差的一半
  - 然后按照从 expired 队列到 active 队列、从高优先级进程到低优先级进程的顺序进行迁移

Linux 2.6负载均衡的“空闲平衡”

- 空闲状态下的负载平衡的调用时机
  - 在调度器中，本 cpu 的就绪队列为空
  - 本cpu的就绪队列为空，且当前绝对时间是 IDLE_REBALANCE_TICK 的倍数，即每隔 IDLE_REBALANCE_TICK 执行一次
- “空闲平衡”的候选进程的标准
  - 与“忙平衡”类似，但因为空闲平衡仅“拉”一个进程过来，动作要小得多，且执行频率相对较高（是忙平衡的 200 倍）
  - 不考虑负载的历史情况和负载差，候选的迁移进程也没有考虑 Cache 活跃程度

load balance()核心代码

```
static int load_balance(int this_cpu, runqueue_t *this_rq,
   struct sched_domain *sd, enum idle_type idle)
{
    struct sched_group *group;
    runqueue_t *busiest;
    unsigned long imbalance;
    int nr_moved;
   spin_lock(&this_rq->lock);
   group = find_busiest_group(sd, this_cpu, &imbalance, idle);
   if (!group)
     goto out_balanced;
  busiest = find_busiest_queue(group);
  if (!busiest)
    goto out_balanced;
```

load balance()核心代码

```
nr_moved = 0;
if (busiest->nr_running > 1) {
  double_lock_balance(this_rq, busiest);
  nr_moved = move_tasks(this_rq, this_cpu, busiest,
    imbalance, sd, idle);
  spin_unlock(&busiest->lock);
}
spin_unlock(&this_rq->lock);
if (!nr_moved) {
  sd->nr_balance_failed++;
  if (unlikely(sd->nr_balance_failed > sd->cache_nice_tries+2)) {
    int wake = 0;
```

load balance()核心代码

```
        spin_lock(&busiest->lock);
        if (!busiest->active_balance) {
                busiest->active_balance = 1;
                busiest->push_cpu = this_cpu;
                wake = 1;
        }
        spin_unlock(&busiest->lock);
        if (wake)
          wake_up_process(busiest->migration_thread);
        sd->nr_balance_failed = sd->cache_nice_tries;
    }
} else
    sd->nr_balance_failed = 0;
```

load balance()核心代码

```
    sd->balance_interval = sd->min_interval;
    return nr_moved;

out_balanced:
    spin_unlock(&this_rq->lock);
    if (sd->balance_interval < sd->max_interval)
      sd->balance_interval *= 2;
    return 0;
}
```

Linux 2.6负载均衡的“推”操作

- 执行体：**migration_thread()**
  - 在系统启动时自动加载（每个 cpu 一个），并将自己设为 SCHED_FIFO 的实时进程
  - 判断标准：检查migration_queue 中是否有请求等待处理
    - ✓ 若没有，就在 TASK_INTERRUPTIBLE 中休眠，直至被唤醒后再次检查
  - migration_queue的添加：**set_cpu_allowed()**
    - ✓ 为待迁移进程构造一个迁移请求数据结构 migration_req_t，将其植入进程所在 cpu 就绪队列的 migration_queue 中
    - ✓ 唤醒该就绪队列的迁移 daemon（记录在 migration_thread 属性中），将该进程迁移到合适的 cpu上去
    - ✓ 目前实现中目的 cpu 的选择和负载无关

主要内容

- 背景知识
  - 调度策略和调度机制
  - Linux 2.4的进程调度机制
  - Linux 2.6的进程调度机制
- 实验内容
  - 将Linux 2.6调度算法修改成随机调度算法

将Linux 2.6调度算法修改成随机调度算法

❖实验说明

- 将把Linux 2.6的调度算法修改成为随机调度算法
- 该算法要求从可运行队列中随机抽取一个进程来运行

❖解决方案

- 修改schedule( )
  - ✓将其选择最高优先级进程的代码修改成随机选择进程
- 对实时进程的考虑
  - ✓对于实时进程，还采用原来的调度方式；而对于普通进程，则采用新的随机调度算法
- 随机数的生成
  - ✓利用jiffies变量来实现伪随机数，因为jiffies变量用于记录时间，每次schedule( )函数被调用时，该变量都很可能发生变化

第15章 进程调度