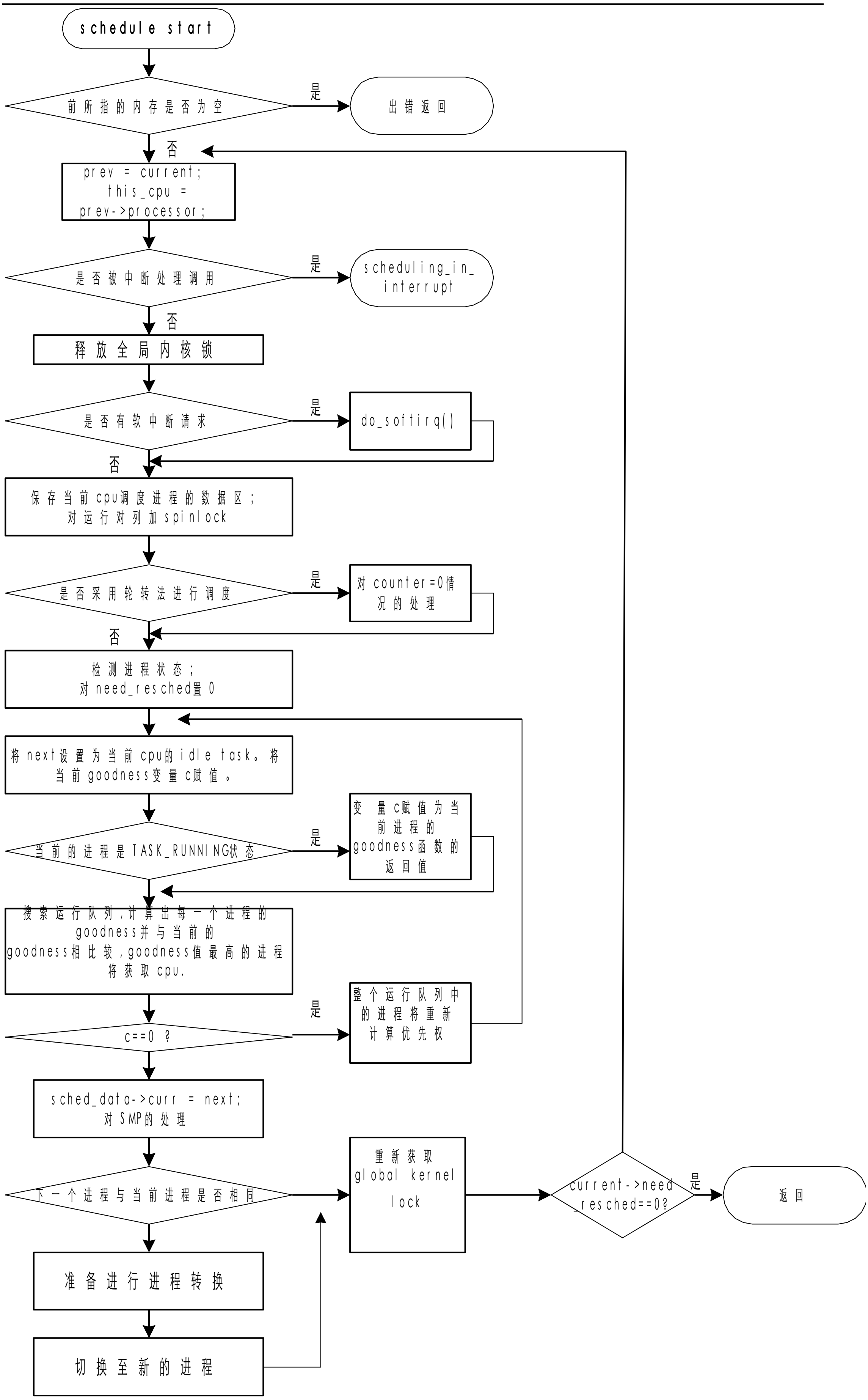
Linux 2.4 版的 schedule() 函数的流程图



kernel/sched.c

```c
/*
 *  'schedule()' is the scheduler function. It's a very simple and nice
 * scheduler: it's not perfect, but certainly works for most things.
 *
 * The goto is "interesting".
 *
 *   NOTE!!  Task 0 is the 'idle' task, which gets called when no other
 * tasks can run. It can not be killed, and it cannot sleep. The 'state'
 * information in task[0] is never used.
 */
asmlinkage void schedule(void)
{
	struct schedule_data * sched_data;
	struct task_struct *prev, *next, *p;
	struct list_head *tmp;
	int this_cpu, c;


	if (!current->active_mm) BUG();
```

如果 current->active_mm == NULL 那么进程必定有问题.任何进程都有一个确定的 active_mm.

```c
need_resched_back:
	prev = current;
	this_cpu = prev->processor;
```

准备交出 CPU 的进程，它的 PCB 由 current 指示。将局部变量 prev 指向当前进程，this_cpu 指向当前进程拥有的 CPU（考虑对称多 CPU 的架构）。

```c

	if (in_interrupt())
		goto scheduling_in_interrupt;
```

检查一下，调用 schedule()函数时是否处于中断处理过程。若是，则打印提示信息后，直接返回。

```c

	release_kernel_lock(prev, this_cpu);
```

释放全局内核锁.

```
/* Do "administrative" work here while we don't hold any locks */
if (softirq_active(this_cpu) & softirq_mask(this_cpu))
        goto handle_softirq;
```

如果有些工作需要由 softirq 机制完成的，那么立刻执行 do_softirq() (kernel/softirq.c)

```
handle_softirq_back:

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

用局部变量 sched_data 保存当前 CPU 的调度进程的数据区。因为任一时刻，每个 CPU 只能运行一个 schedule()。schedule_data 结构包含一个指向当前进程 task_struct 结构的指针和最后一个调度进程的 TSC 值(Time Stamp Counter，CPU 附带一个 64 位的时间戳寄存器)。

```
spin_lock_irq(&runqueue_lock);
```

对就绪队列加 spinlock。之所以用 spinlock，是因为 schedule()是允许中断的。这样在就绪队列解锁之后，我们可以直接使中断有效，而不必做一些保存标志、恢复标志的操作。

```
/* move an exhausted RR process to be last.. */
if (prev->policy == SCHED_RR)
        goto move_rr_last;
```

如果采用轮转法进行调度，则要重新检查一下 counter 是否为 0。若是 0，则将其挂到就绪队列的最后。

```
move_rr_back:

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

```
		default:
			del_from_runqueue(prev);
		case TASK_RUNNING:
	}
```

检测进程状态：如果是 TASK_RUNNING 则暂且放在一边不管；如果是 TASK_INTERRUPTIBLE 状态并且能够唤醒它的信号已经迫近，则将其状态置为 TASK_RUNNING。至于是别的状态，则将其从就绪队列中删去。

```
	prev->need_resched = 0;
```

对 need_resched 置 0，使下个时间片不会有新的 schedule()函数调用。

```
	/*
	 * this is the scheduler proper:
	 */
repeat_schedule:
	/*
	 * Default process to select..
	 */
	next = idle_task(this_cpu);
	c = -1000;
```

将 next 设置为当前 cpu 的 idle task。将当前 goodness(优先级)变量 c 赋值-1000。 -1000 是 goodness 函数返回值之中最小的权。

```
	if (prev->state == TASK_RUNNING)
		goto still_running;
```

如果当前的进程是 TASK_RUNNING 状态，则将当前 goodness(优先级)变量 c 赋值为当前进程的 goodness 函数的返回值，这将优于 idle task。

```
still_running_back:
	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;
		}
	}
```

搜索就绪队列，计算出每一个进程的 goodness(优先级)并与当前的 goodness(优先级)相比较。goodness(优先级)值最高的进程将获取 CPU。

```
	/* Do we need to re-calculate counters? */
	if (!c)
		goto recalculate;
```

如果整个就绪队列中最高的 goodness(优先级)值是 0，那么整个运行队列中的进程将重新计算优先权。算法如下：

```
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);
```

```
	/*
	 * from this point on nothing can prevent us from
	 * switching to the next task, save this fact in
	 * sched_data.
	 */
	sched_data->curr = next;
```

从这一时刻起，要转换到下一个任务，将当前信息保存到 sched_data 中。

```
#ifdef CONFIG_SMP
	next->has_cpu = 1;
	next->processor = this_cpu;
#endif
```

next 进程拥有一个 CPU，它就是 this_cpu

```
	spin_unlock_irq(&runqueue_lock);

	if (prev == next)
		goto same_process;
```

处理一种特殊情况，可能选中的 next 进程就是当前进程。那么，我们只需重新获取 global kernel lock 并返回。

```
#ifdef CONFIG_SMP
```

```
	/*
	 * maintain the per-process 'last schedule' value.
	 * (this has to be recalculated even if we reschedule to
	 * the same process) Currently this is only used on SMP,
	 * and it's approximate, so we do not have to maintain
	 * it while holding the runqueue spinlock.
	 */
	sched_data->last_schedule = get_cycles();

	/*
	 * We drop the scheduler lock early (it's a global spinlock),
	 * thus we have to lock the previous process from getting
	 * rescheduled during switch_to().
	 */

#endif /* CONFIG_SMP */

	kstat.context_swtch++;
```

累计上下文切换次数递增 1

```
	/*
	 * there are 3 processes which are affected by a context switch:
	 *
	 * prev == .... ==> (last => next)
	 *
	 * It's the 'much more previous' 'prev' that is on next's stack,
	 * but prev is set to (the just run) 'last' process by switch_to().
	 * This might sound slightly confusing but makes tons of sense.
	 */
	prepare_to_switch();
	{
		struct mm_struct *mm = next->mm;
		struct mm_struct *oldmm = prev->active_mm;
		if (!mm) {
			if (next->active_mm) BUG();
			next->active_mm = oldmm;
			atomic_inc(&oldmm->mm_count);
			enter_lazy_tlb(oldmm, next, this_cpu);
		} else {
```

```
                if (next->active_mm != mm) BUG();
                switch_mm(oldmm, mm, next, this_cpu);
        }
```

切换内存管理数据结构，从 prev 至 next

```

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

如果 prev 本来就共享了其它进程的 mm，那么，active_mm 清 0，oldmm 的共享计数递减 1。

```
    }

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

切换寄存器和堆栈。从而，进程切换全部完成！

```
    __schedule_tail(prev);
```

这次切换后失去 CPU 的 prev，仍然留在就绪队列。它终究会重新获得 CPU。一旦重新获得，其 PC 寄存器指向现在这个位置，也即_schedule_tail(prev)。

```

same_process:
```

选中的 next 进程就是当前进程

```
    reacquire_kernel_lock(current);
    if (current->need_resched)
            goto need_resched_back;
```

如果又有新一轮的调度需求，那么转 need_resched_back 去调度。

```

    return;
```

否则，schedule()函数返回了 !!!

```

recalculate:
```

重新计算就绪队列中所有进程的优先权。前文已有说明

```
	{
		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;
```

重新计算就绪队列中所有进程的优先权。前文已有说明

```

still_running:
	c = goodness(prev, this_cpu, prev->active_mm);
	next = prev;
	goto still_running_back;
```

如果 prev 本身也是就绪进程，那么，先计算 prev 的优先权，先把 prev 设置成 next。后续会逐个计算机就绪队列里面之进程的优先权，如果有更高优先权的进程，会重置 c 和 next。

```

handle_softirq:
	do_softirq();
	goto handle_softirq_back;

move_rr_last:
	if (!prev->counter) {
		prev->counter = NICE_TO_TICKS(prev->nice);
		move_last_runqueue(prev);
	}
	goto move_rr_back;
```

更新 prev 的优先权，并且把它放置就绪队列的末尾。

```

scheduling_in_interrupt:
	printk("Scheduling in interrupt\n");
	BUG();
	return;
}
```

Linux 2.4 版的 goodness() 函数

进程的 goodness 值通过 goodness()函数计算。goodness 返回下面两类中的一个值：

- 1000 和 1000 以上的值只能赋给实时进程，
- 从 0-999 的值只能赋给普通进程。

有关这两类 goodness 结果的重要的一点是：该值在实时系统的范围肯定会比非实时系统的范围要高。POSIX 标准规定内核要确保在实时进程和非实时进程同时竞争 CPU 时，实时进程要优先于非实时进程。由于调度进程总是选择具有最大 goodness 值的进程，又由于任何尚未释放 CPU 的实时进程的 goodness 值总是比非实时进程的 goodness 大，Linux 是遵守 POSIX 标准的。

goodness()函数从不会返回-1000 的，也不会返回其它的负值。

而 idle 进程的 counter 值为负。

kernel/sched.c

```
/*
 * This is the function that decides how desirable a process is..
 * You can weigh different processes against each other depending
 * on what CPU they've run on lately etc to try to handle cache
 * and TLB miss penalties.
 *
 * Return values:
 *      -1000: never select this
 *          0: out of time, recalculate counters (but it might still be
 *             selected)
 *        +ve: "goodness" value (the larger, the better)
 *      +1000: realtime process, select this.
 */

static inline int goodness(struct task_struct * p, int this_cpu, struct mm_struct *this_mm)
{
        int weight;

        /*
         * select the current process after every other
         * runnable process, but before the idle thread.
```

```
         * Also, dont trigger a counter recalculation.
         */
        weight = -1;
        if (p->policy & SCHED_YIELD)
                goto out;
```

如果要放弃这次调度，那么直接返回。

```

        /*
         * Non-RT process - normal case first.
         */
        if (p->policy == SCHED_OTHER) {
```

如果是非实时进程调度，执行下列控制流

```
                /*
                 * Give the process a first-approximation goodness value
                 * according to the number of clock-ticks it has left.
                 *
                 * Don't do any other calculations if the time slice is
                 * over..
                 */
                weight = p->counter;
```

时间片剩余值作为优先权。

```
                if (!weight)
                        goto out;
```

若时间片用完，则返回 0。

```

#ifdef CONFIG_SMP
                /* Give a largish advantage to the same processor...   */
                /* (this is equivalent to penalizing other processors) */
                if (p->processor == this_cpu)
                        weight += PROC_CHANGE_PENALTY;
#endif

                /* .. and a slight advantage to the current MM */
                if (p->mm == this_mm || !p->mm)
                        weight += 1;
```

如果进程的存储区与当前的一致，或者进程是内核线程，则优先权增1。

```
		weight += 20 - p->nice;
```

最后，根据p进程的nice值适当地调整优先权。

```
		goto out;
	}

	/*
	 * Realtime process, select the first one on the
	 * runqueue (taking priorities within processes
	 * into account).
	 */
	weight = 1000 + p->rt_priority;
```

如果是实时进程调度，则取得实时进程优先权，而不是counter。加上1000，使其优先权高于任何普通进程。

```
out:
	return weight;
}
```