```
// Code not shown here decreases the current process's
// counter once every tick (10ms on a PC). When it reaches
// 0, the procedure schedule() in this file is invoked to
// choose the next process to run.  This choice may also
// happen earlier if the current process needs to wait
// for something, or if a non-runnable process becomes
// runnable.  (The decision to reschedule immediately in
// the latter case is made by code in reschedule_idle().)
```

```
/*
 *  linux/kernel/sched.c
 *
 *  Kernel scheduler and related syscalls
 *
 *  Copyright (C) 1991, 1992  Linus Torvalds
 *
 *  1996-12-23  Modified by Dave Grothe to fix bugs in semaphores and
 *              make semaphores SMP safe
 *  1998-11-19  Implemented schedule_timeout() and related stuff
 *              by Andrea Arcangeli
 *  1998-12-28  Implemented better SMP scheduling by Ingo Molnar
 */

/*
 * 'sched.c' is the main kernel file. It contains scheduling primitives
 * (sleep_on, wakeup, schedule etc) as well as a number of simple system
 * call functions (type getpid()), which just extract a field from
 * current-task
 */

#include <linux/config.h>
#include <linux/mm.h>
#include <linux/init.h>
#include <linux/smp_lock.h>
#include <linux/interrupt.h>
#include <linux/kernel_stat.h>

#include <asm/uaccess.h>
#include <asm/mmu_context.h>

extern void timer_bh(void);
extern void tqueue_bh(void);
extern void immediate_bh(void);

/*
 * scheduler variables
 */

unsigned securebits = SECUREBITS_DEFAULT; /* systemwide security settings */

extern void mem_use(void);

/*
 * Scheduling quanta.
 *
 * NOTE! The unix "nice" value influences how long a process
 * gets. The nice value ranges from -20 to +19, where a -20
 * is a "high-priority" task, and a "+10" is a low-priority
 * task.
 *
 * We want the time-slice to be around 50ms or so, so this
 * calculation depends on the value of HZ.
 */
#if HZ < 200
#define TICK_SCALE(x)   ((x) >> 2)
#elif HZ < 400
#define TICK_SCALE(x)   ((x) >> 1)
#elif HZ < 800
#define TICK_SCALE(x)   (x)
#elif HZ < 1600
#define TICK_SCALE(x)   ((x) << 1)
#else
#define TICK_SCALE(x)   ((x) << 2)
#endif
```

```
// On a PC, HZ = 100, so the first version of TICK_SCALE
// is used.  The C operation ">> 2" means to shift right
// two bit positions, i.e., take the quotient of division
// by 4.
```

```
#define NICE_TO_TICKS(nice)     (TICK_SCALE(20-(nice))+1)                                           // So a process gets 1 extra tick per 4 niceness points.


/*
 *      Init task must be ok at boot for the ix86 as we will check its signals
 *      via the SMP irq return path.
 */

struct task_struct * init_tasks[NR_CPUS] = {&init_task, };

/*
 * The tasklist_lock protects the linked list of processes.
 *
 * The scheduler lock is protecting against multiple entry
 * into the scheduling code, and doesn't need to worry
 * about interrupts (because interrupts cannot call the
 * scheduler).
 *
 * The run-queue lock locks the parts that actually access
 * and change the run-queues, and have to be interrupt-safe.
 */
spinlock_t runqueue_lock __cacheline_aligned = SPIN_LOCK_UNLOCKED;  /* second */
rwlock_t tasklist_lock __cacheline_aligned = RW_LOCK_UNLOCKED;  /* third */

static LIST_HEAD(runqueue_head);

/*
 * We align per-CPU scheduling data on cacheline boundaries,
 * to prevent cacheline ping-pong.
 */
static union {
        struct schedule_data {
                struct task_struct * curr;
                cycles_t last_schedule;
        } schedule_data;
        char __pad [SMP_CACHE_BYTES];
} aligned_data [NR_CPUS] __cacheline_aligned = { {{&init_task,0}}};

#define cpu_curr(cpu) aligned_data[(cpu)].schedule_data.curr
#define last_schedule(cpu) aligned_data[(cpu)].schedule_data.last_schedule

struct kernel_stat kstat;

#ifdef CONFIG_SMP

#define idle_task(cpu) (init_tasks[cpu_number_map(cpu)])
#define can_schedule(p,cpu) ((!(p)->has_cpu) && \
                                ((p)->cpus_allowed & (1 << cpu)))

#else

#define idle_task(cpu) (&init_task)
#define can_schedule(p,cpu) (1)

#endif

void scheduling_functions_start_here(void) { }

/*
 * 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;

        /*
         * Realtime process, select the first one on the
         * runqueue (taking priorities within processes
         * into account).
         */
        if (p->policy != SCHED_OTHER) {
                weight = 1000 + p->rt_priority;
                goto out;
        }

        /*
         * 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;                                                                   // This is one of the two normal components of the goodness:
        if (!weight)                                                                           // how many ticks are left in the counter.
                goto out;

#ifdef CONFIG_SMP                                                                              // This block (until #endif) can be ignored since we don't
        /* Give a largish advantage to the same processor...   */                              // care about SMP systems.
        /* (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 */                                      // Normally this just favors the current process; it also
        if (p->mm == this_mm || !p->mm)                                                        //    favors any other process sharing the same memory
                weight += 1;                                                                   //    nmapping, but that is irrelevant for our experiments.
        weight += 20 - p->nice;                                                                // This is the other normal component of the goodness:
                                                                                               //    the process's niceness (subtracted from 20 so higher
out:                                                                                           //    priority = numerically higher).
        return weight;
}

/*
 * subtle. We want to discard a yielded process only if it's being
 * considered for a reschedule. Wakeup-time 'queries' of the scheduling
 * state do not count. Another optimization we do: sched_yield()-ed
 * processes are runnable (and thus will be considered for scheduling)
 * right when they are calling schedule(). So the only place we need
 * to care about SCHED_YIELD is when we calculate the previous process'
 * goodness ...
 */
static inline int prev_goodness(struct task_struct * p, int this_cpu, struct mm_struct *this_mm)
{
        if (p->policy & SCHED_YIELD) {
                p->policy &= ~SCHED_YIELD;
                return 0;
        }
        return goodness(p, this_cpu, this_mm);
}

/*
```

```
 * the 'goodness value' of replacing a process on a given CPU.
 * positive value means 'replace', zero or negative means 'dont'.
 */
static inline int preemption_goodness(struct task_struct * prev, struct task_struct * p, int cpu)
{
        return goodness(p, cpu, prev->active_mm) - goodness(prev, cpu, prev->active_mm);
}

/*
 * This is ugly, but reschedule_idle() is very timing-critical.
 * We enter with the runqueue spinlock held, but we might end
 * up unlocking it early, so the caller must not unlock the
 * runqueue, it's always done by reschedule_idle().
 *
 * This function must be inline as anything that saves and restores
 * flags has to do so within the same register window on sparc (Anton)
 */
static inline void reschedule_idle(struct task_struct * p, unsigned long flags)
{
#ifdef CONFIG_SMP                                                                         // The huge block until #else is irrelevant for us, since
        int this_cpu = smp_processor_id();                                                // we are only using uniprocessor systems.
        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)
                        goto send_now_idle;

                /*
                 * Maybe this process has enough priority to preempt
                 * its preferred CPU. (this is a shortcut):
                 */
                tsk = cpu_curr(best_cpu);
                if (preemption_goodness(tsk, p, best_cpu) > 0)
                        goto preempt_now;
        }

        /*
         * 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 = 1;

        for (i = 0; i < smp_num_cpus; i++) {
                cpu = cpu_logical_map(i);
                if (!can_schedule(p, cpu))
                        continue;
                tsk = cpu_curr(cpu);
                /*
                 * 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 (last_schedule(cpu) < oldest_idle) {
                                oldest_idle = last_schedule(cpu);
                                target_tsk = tsk;
                        }
                } else {
                        if (oldest_idle == -1ULL) {
                                int prio = preemption_goodness(tsk, p, cpu);

                                if (prio > max_prio) {
                                        max_prio = prio;
                                        target_tsk = tsk;
                                }
                        }
                }
        }
        tsk = target_tsk;
        if (tsk) {
                if (oldest_idle != -1ULL)
                        goto send_now_idle;
                goto preempt_now;
        }

        spin_unlock_irqrestore(&runqueue_lock, flags);
        return;

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.
         */
        if ((tsk->processor != current->processor) && !tsk->need_resched)
                smp_send_reschedule(tsk->processor);
        tsk->need_resched = 1;
        spin_unlock_irqrestore(&runqueue_lock, flags);
        return;

preempt_now:
        tsk->need_resched = 1;
        spin_unlock_irqrestore(&runqueue_lock, flags);
        /*
         * the APIC stuff can go outside of the lock because
         * it uses no task information, only CPU#.
         */
        if (tsk->processor != this_cpu)
                smp_send_reschedule(tsk->processor);
        return;
#else /* UP */
        int this_cpu = smp_processor_id();                                         // This is done when a processs (p) becomes runnable.
        struct task_struct *tsk;

        tsk = cpu_curr(this_cpu);
        if (preemption_goodness(tsk, p, this_cpu) > 1)                             // Is the newly runanble process enough bettter than the
                tsk->need_resched = 1;                                             // currently running process to be worth immediately
        spin_unlock_irqrestore(&runqueue_lock, flags);                             // rescheduling, rather than waiting?  Note that even
#endif                                                                             // if the answer is yes, the same process could get
}                                                                                  // chosen to run again (or the new one, or any other).

/*
 * Careful!                                                                        // This comment is obsolete.
 *
 * This has to add the process to the _beginning_ of the
 * run-queue, not the end. See the comment about "This is
 * subtle" in the scheduler proper..
 */
static inline void add_to_runqueue(struct task_struct * p)
```

```
{
        list_add(&p->run_list, &runqueue_head);                                                      // To see which end of the queue this is adding to, compare
        nr_running++;                                                                                 // it with the next two procedures.
}

static inline void move_last_runqueue(struct task_struct * p)
{
        list_del(&p->run_list);
        list_add_tail(&p->run_list, &runqueue_head);
}

static inline void move_first_runqueue(struct task_struct * p)
{
        list_del(&p->run_list);
        list_add(&p->run_list, &runqueue_head);
}

/*
 * Wake up a process. Put it on the run-queue if it's not
 * already there.  The "current" process is always on the
 * run-queue (except when the actual re-schedule is in
 * progress), and as such you're allowed to do the simpler
 * "current->state = TASK_RUNNING" to mark yourself runnable
 * without the overhead of this.
 */
inline void wake_up_process(struct task_struct * p)
{
        unsigned long flags;

        /*
         * We want the common case fall through straight, thus the goto.
         */
        spin_lock_irqsave(&runqueue_lock, flags);
        p->state = TASK_RUNNING;
        if (task_on_runqueue(p))
                goto out;
        add_to_runqueue(p);
        reschedule_idle(p, flags); // spin_unlocks runqueue

        return;
out:
        spin_unlock_irqrestore(&runqueue_lock, flags);
}

static inline void wake_up_process_synchronous(struct task_struct * p)
{
        unsigned long flags;

        /*
         * We want the common case fall through straight, thus the goto.
         */
        spin_lock_irqsave(&runqueue_lock, flags);
        p->state = TASK_RUNNING;
        if (task_on_runqueue(p))
                goto out;
        add_to_runqueue(p);
out:
        spin_unlock_irqrestore(&runqueue_lock, flags);
}

static void process_timeout(unsigned long __data)
{
        struct task_struct * p = (struct task_struct *) __data;

        wake_up_process(p);
}
```

```
signed long schedule_timeout(signed long timeout)
{
        struct timer_list timer;
        unsigned long expire;

        switch (timeout)
        {
        case MAX_SCHEDULE_TIMEOUT:
                /*
                 * These two special cases are useful to be comfortable
                 * in the caller. Nothing more. We could take
                 * MAX_SCHEDULE_TIMEOUT from one of the negative value
                 * but I' d like to return a valid offset (>=0) to allow
                 * the caller to do everything it want with the retval.
                 */
                schedule();
                goto out;
        default:
                /*
                 * Another bit of PARANOID. Note that the retval will be
                 * 0 since no piece of kernel is supposed to do a check
                 * for a negative retval of schedule_timeout() (since it
                 * should never happens anyway). You just have the printk()
                 * that will tell you if something is gone wrong and where.
                 */
                if (timeout < 0)
                {
                        printk(KERN_ERR "schedule_timeout: wrong timeout "
                               "value %lx from %p\n", timeout,
                               __builtin_return_address(0));
                        current->state = TASK_RUNNING;
                        goto out;
                }
        }

        expire = timeout + jiffies;

        init_timer(&timer);
        timer.expires = expire;
        timer.data = (unsigned long) current;
        timer.function = process_timeout;

        add_timer(&timer);
        schedule();
        del_timer_sync(&timer);

        timeout = expire - jiffies;

 out:
        return timeout < 0 ? 0 : timeout;
}

/*
 * schedule_tail() is getting called from the fork return path. This
 * cleans up all remaining scheduler things, without impacting the
 * common case.
 */
static inline void __schedule_tail(struct task_struct *prev)
{
#ifdef CONFIG_SMP
        if ((prev->state == TASK_RUNNING) &&
                        (prev != idle_task(smp_processor_id()))) {
                unsigned long flags;

                spin_lock_irqsave(&runqueue_lock, flags);
```

```
                prev->has_cpu = 0;
                reschedule_idle(prev, flags); // spin_unlocks runqueue
        } else {
                wmb();
                prev->has_cpu = 0;
        }
#endif /* CONFIG_SMP */
}

void schedule_tail(struct task_struct *prev)
{
        __schedule_tail(prev);
}

/*
 *  '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;                                                      // You can ignore everything until the comment that says
        struct task_struct *prev, *next, *p;                                                    // "this is the scheduler proper".
        struct list_head *tmp;
        int this_cpu, c;

        if (!current->active_mm) BUG();
        if (tq_scheduler)
                goto handle_tq_scheduler;
tq_scheduler_back:

        prev = current;
        this_cpu = prev->processor;

        if (in_interrupt())
                goto scheduling_in_interrupt;

        release_kernel_lock(prev, this_cpu);

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

        spin_lock_irq(&runqueue_lock);

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

        switch (prev->state & ~TASK_EXCLUSIVE) {
                case TASK_INTERRUPTIBLE:
                        if (signal_pending(prev)) {
                                prev->state = TASK_RUNNING;
```

```
                                break;
                        }
                default:
                        del_from_runqueue(prev);
                case TASK_RUNNING:
        }
        prev->need_resched = 0;

        /*
         * this is the scheduler proper:                                                          // "next" is the next process to run, "c" is its goodness
         */                                                                                       // These are used in a "king of the hill" fashion, keeping
                                                                                                  // track of the best choice found so far.
repeat_schedule:
        /*                                                                                        // If the current task is no longer runnable, we'll take a
         * Default process to select..                                                            //   so-called idle task as the initial king (best so far).
         */                                                                                       //   The idle task does no useful work, so we want it to
        next = idle_task(this_cpu);                                                               //   win only if there are no other competitors; hence we
        c = -1000;                                                                                //   give it a lousy score of -1000; anybody will beat it.
        if (prev->state == TASK_RUNNING)                                                          // If on the other hand the current (previous) task is still
                goto still_running;                                                               //   runnable, go to code that initializes differently.

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)                                                           // Update the best found so far.  Note what happens if there
                                c = weight, next = p;                                             // is a tie.
                }
        }

        /* Do we need to re-calculate counters? */
        if (!c)                                                                                   // If all runnable processes' counters are 0, charge the
          goto recalculate;                                                                       // counters back up (and repeat the scheduling loop).
        /*
         * from this point on nothing can prevent us from
         * switching to the next task, save this fact in
         * sched_data.
         */
        sched_data->curr = next;
#ifdef CONFIG_SMP
        next->has_cpu = 1;
        next->processor = this_cpu;
#endif
        spin_unlock_irq(&runqueue_lock);

        if (prev == next)
                goto same_process;

#ifdef CONFIG_SMP
        /*
         * maintain the per-process 'average timeslice' 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.
         */
        {
                cycles_t t, this_slice;

                t = get_cycles();
                this_slice = t - sched_data->last_schedule;
                sched_data->last_schedule = t;
        }

        /*
```

```
         * 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++;
        /*
         * 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);
                }

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

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

same_process:
        reacquire_kernel_lock(current);
        return;

recalculate:                                                                                         // Charge counters back up -- for *all* processes, not just
        {                                                                                            // the runnable ones (which all reached 0).
                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);                    // Note that if the counter isn't 0, half of the old value
                read_unlock(&tasklist_lock);                                                         // is retained (because >> 1 is shift by 1 bit, i.e.,
                spin_lock_irq(&runqueue_lock);                                                       // take the quotient by 2).
        }
        goto repeat_schedule;

still_running:
        c = prev_goodness(prev, this_cpu, prev->active_mm);                                          // Initial "king" (best so far) if current process is
        next = prev;                                                                                 // runnable is that current (i.e., previous) process.
        goto still_running_back;                                                                     // For our purposes, prev_goodness is the same as
                                                                                                     // goodness.
handle_softirq:
```

```
	do_softirq();
	goto handle_softirq_back;

handle_tq_scheduler:
	/*
	 * do not run the task queue with disabled interrupts,
	 * cli() wouldn't work on SMP
	 */
	sti();
	run_task_queue(&tq_scheduler);
	goto tq_scheduler_back;

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

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

static inline void __wake_up_common (wait_queue_head_t *q, unsigned int mode,
						const int sync)
{
	struct list_head *tmp, *head;
	struct task_struct *p, *best_exclusive;
	unsigned long flags;
	int best_cpu, irq;

	if (!q)
		goto out;

	best_cpu = smp_processor_id();
	irq = in_interrupt();
	best_exclusive = NULL;
	wq_write_lock_irqsave(&q->lock, flags);

#if WAITQUEUE_DEBUG
	CHECK_MAGIC_WQHEAD(q);
#endif

	head = &q->task_list;
#if WAITQUEUE_DEBUG
	if (!head->next || !head->prev)
		WQ_BUG();
#endif
	tmp = head->next;
	while (tmp != head) {
		unsigned int state;
		wait_queue_t *curr = list_entry(tmp, wait_queue_t, task_list);

		tmp = tmp->next;

#if WAITQUEUE_DEBUG
		CHECK_MAGIC(curr->__magic);
#endif
		p = curr->task;
		state = p->state;
		if (state & (mode & ~TASK_EXCLUSIVE)) {
#if WAITQUEUE_DEBUG
			curr->__waker = (long)__builtin_return_address(0);
#endif
```

```
                        /*
                         * If waking up from an interrupt context then
                         * prefer processes which are affine to this
                         * CPU.
                         */
                        if (irq && (state & mode & TASK_EXCLUSIVE)) {
                                if (!best_exclusive)
                                        best_exclusive = p;
                                else if ((p->processor == best_cpu) &&
                                         (best_exclusive->processor != best_cpu))
                                                best_exclusive = p;
                        } else {
                                if (sync)
                                        wake_up_process_synchronous(p);
                                else
                                        wake_up_process(p);
                                if (state & mode & TASK_EXCLUSIVE)
                                        break;
                        }
                }
        }
        if (best_exclusive)
                best_exclusive->state = TASK_RUNNING;
        wq_write_unlock_irqrestore(&q->lock, flags);

        if (best_exclusive) {
                if (sync)
                        wake_up_process_synchronous(best_exclusive);
                else
                        wake_up_process(best_exclusive);
        }
out:
        return;
}

void __wake_up(wait_queue_head_t *q, unsigned int mode)
{
        __wake_up_common(q, mode, 0);
}

void __wake_up_sync(wait_queue_head_t *q, unsigned int mode)
{
        __wake_up_common(q, mode, 1);
}

#define SLEEP_ON_VAR                            \
        unsigned long flags;                    \
        wait_queue_t wait;                      \
        init_waitqueue_entry(&wait, current);

#define SLEEP_ON_HEAD                                   \
        wq_write_lock_irqsave(&q->lock,flags);          \
        __add_wait_queue(q, &wait);                     \
        wq_write_unlock(&q->lock);

#define SLEEP_ON_TAIL                                           \
        wq_write_lock_irq(&q->lock);                            \
        __remove_wait_queue(q, &wait);                          \
        wq_write_unlock_irqrestore(&q->lock,flags);

void interruptible_sleep_on(wait_queue_head_t *q)
{
        SLEEP_ON_VAR

        current->state = TASK_INTERRUPTIBLE;
```

```
        SLEEP_ON_HEAD
        schedule();
        SLEEP_ON_TAIL
}

long interruptible_sleep_on_timeout(wait_queue_head_t *q, long timeout)
{
        SLEEP_ON_VAR

        current->state = TASK_INTERRUPTIBLE;

        SLEEP_ON_HEAD
        timeout = schedule_timeout(timeout);
        SLEEP_ON_TAIL

        return timeout;
}

void sleep_on(wait_queue_head_t *q)
{
        SLEEP_ON_VAR

        current->state = TASK_UNINTERRUPTIBLE;

        SLEEP_ON_HEAD
        schedule();
        SLEEP_ON_TAIL
}

long sleep_on_timeout(wait_queue_head_t *q, long timeout)
{
        SLEEP_ON_VAR

        current->state = TASK_UNINTERRUPTIBLE;

        SLEEP_ON_HEAD
        timeout = schedule_timeout(timeout);
        SLEEP_ON_TAIL

        return timeout;
}

void scheduling_functions_end_here(void) { }

#ifndef __alpha__

/*
 * This has been replaced by sys_setpriority.  Maybe it should be
 * moved into the arch dependent tree for those ports that require
 * it for backward compatibility?
 */

asmlinkage long sys_nice(int increment)
{
        long newprio;

        /*
         *      Setpriority might change our priority at the same moment.
         *      We don't have to worry. Conceptually one call occurs first
         *      and we have a single winner.
         */
        if (increment < 0) {
                if (!capable(CAP_SYS_NICE))
                        return -EPERM;
                if (increment < -40)
                        increment = -40;
```

```
	}
	if (increment > 40)
		increment = 40;

	newprio = current->nice + increment;
	if (newprio < -20)
		newprio = -20;
	if (newprio > 19)
		newprio = 19;
	current->nice = newprio;
	return 0;
}

#endif

static inline struct task_struct *find_process_by_pid(pid_t pid)
{
	struct task_struct *tsk = current;

	if (pid)
		tsk = find_task_by_pid(pid);
	return tsk;
}

static int setscheduler(pid_t pid, int policy,
			struct sched_param *param)
{
	struct sched_param lp;
	struct task_struct *p;
	int retval;

	retval = -EINVAL;
	if (!param || pid < 0)
		goto out_nounlock;

	retval = -EFAULT;
	if (copy_from_user(&lp, param, sizeof(struct sched_param)))
		goto out_nounlock;

	/*
	 * We play safe to avoid deadlocks.
	 */
	spin_lock_irq(&runqueue_lock);
	read_lock(&tasklist_lock);

	p = find_process_by_pid(pid);

	retval = -ESRCH;
	if (!p)
		goto out_unlock;

	if (policy < 0)
		policy = p->policy;
	else {
		retval = -EINVAL;
		if (policy != SCHED_FIFO && policy != SCHED_RR &&
				policy != SCHED_OTHER)
			goto out_unlock;
	}

	/*
	 * Valid priorities for SCHED_FIFO and SCHED_RR are 1..99, valid
	 * priority for SCHED_OTHER is 0.
	 */
	retval = -EINVAL;
	if (lp.sched_priority < 0 || lp.sched_priority > 99)
```

```
                goto out_unlock;
        if ((policy == SCHED_OTHER) != (lp.sched_priority == 0))
                goto out_unlock;

        retval = -EPERM;
        if ((policy == SCHED_FIFO || policy == SCHED_RR) &&
            !capable(CAP_SYS_NICE))
                goto out_unlock;
        if ((current->euid != p->euid) && (current->euid != p->uid) &&
            !capable(CAP_SYS_NICE))
                goto out_unlock;

        retval = 0;
        p->policy = policy;
        p->rt_priority = lp.sched_priority;
        if (task_on_runqueue(p))
                move_first_runqueue(p);

        current->need_resched = 1;

out_unlock:
        read_unlock(&tasklist_lock);
        spin_unlock_irq(&runqueue_lock);

out_nounlock:
        return retval;
}

asmlinkage long sys_sched_setscheduler(pid_t pid, int policy,
                                      struct sched_param *param)
{
        return setscheduler(pid, policy, param);
}

asmlinkage long sys_sched_setparam(pid_t pid, struct sched_param *param)
{
        return setscheduler(pid, -1, param);
}

asmlinkage long sys_sched_getscheduler(pid_t pid)
{
        struct task_struct *p;
        int retval;

        retval = -EINVAL;
        if (pid < 0)
                goto out_nounlock;

        retval = -ESRCH;
        read_lock(&tasklist_lock);
        p = find_process_by_pid(pid);
        if (p)
                retval = p->policy & ~SCHED_YIELD;
        read_unlock(&tasklist_lock);

out_nounlock:
        return retval;
}

asmlinkage long sys_sched_getparam(pid_t pid, struct sched_param *param)
{
        struct task_struct *p;
        struct sched_param lp;
        int retval;

        retval = -EINVAL;
```

```
	if (!param || pid < 0)
		goto out_nounlock;

	read_lock(&tasklist_lock);
	p = find_process_by_pid(pid);
	retval = -ESRCH;
	if (!p)
		goto out_unlock;
	lp.sched_priority = p->rt_priority;
	read_unlock(&tasklist_lock);

	/*
	 * This one might sleep, we cannot do it with a spinlock held ...
	 */
	retval = copy_to_user(param, &lp, sizeof(*param)) ? -EFAULT : 0;

out_nounlock:
	return retval;

out_unlock:
	read_unlock(&tasklist_lock);
	return retval;
}

asmlinkage long sys_sched_yield(void)
{
	spin_lock_irq(&runqueue_lock);
	if (current->policy == SCHED_OTHER)
		current->policy |= SCHED_YIELD;
	current->need_resched = 1;
	move_last_runqueue(current);
	spin_unlock_irq(&runqueue_lock);
	return 0;
}

asmlinkage long sys_sched_get_priority_max(int policy)
{
	int ret = -EINVAL;

	switch (policy) {
	case SCHED_FIFO:
	case SCHED_RR:
		ret = 99;
		break;
	case SCHED_OTHER:
		ret = 0;
		break;
	}
	return ret;
}

asmlinkage long sys_sched_get_priority_min(int policy)
{
	int ret = -EINVAL;

	switch (policy) {
	case SCHED_FIFO:
	case SCHED_RR:
		ret = 1;
		break;
	case SCHED_OTHER:
		ret = 0;
	}
	return ret;
}
```

```
asmlinkage long sys_sched_rr_get_interval(pid_t pid, struct timespec *interval)
{
	struct timespec t;
	struct task_struct *p;
	int retval = -EINVAL;

	if (pid < 0)
		goto out_nounlock;

	retval = -ESRCH;
	read_lock(&tasklist_lock);
	p = find_process_by_pid(pid);
	if (p)
		jiffies_to_timespec(p->policy & SCHED_FIFO ? 0 : NICE_TO_TICKS(p->nice),
				    &t);
	read_unlock(&tasklist_lock);
	if (p)
		retval = copy_to_user(interval, &t, sizeof(t)) ? -EFAULT : 0;
out_nounlock:
	return retval;
}

static void show_task(struct task_struct * p)
{
	unsigned long free = 0;
	int state;
	static const char * stat_nam[] = { "R", "S", "D", "Z", "T", "W" };

	printk("%-8s  ", p->comm);
	state = p->state ? ffz(~p->state) + 1 : 0;
	if (((unsigned) state) < sizeof(stat_nam)/sizeof(char *))
		printk(stat_nam[state]);
	else
		printk(" ");
#if (BITS_PER_LONG == 32)
	if (p == current)
		printk(" current  ");
	else
		printk(" %08lX ", thread_saved_pc(&p->thread));
#else
	if (p == current)
		printk("   current task   ");
	else
		printk(" %016lx ", thread_saved_pc(&p->thread));
#endif
	{
		unsigned long * n = (unsigned long *) (p+1);
		while (!*n)
			n++;
		free = (unsigned long) n - (unsigned long)(p+1);
	}
	printk("%5lu %5d %6d ", free, p->pid, p->p_pptr->pid);
	if (p->p_cptr)
		printk("%5d ", p->p_cptr->pid);
	else
		printk("      ");
	if (!p->mm)
		printk(" (L-TLB) ");
	else
		printk(" (NOTLB) ");
	if (p->p_ysptr)
		printk("%7d", p->p_ysptr->pid);
	else
		printk("       ");
	if (p->p_osptr)
		printk(" %5d\n", p->p_osptr->pid);
```

```
	else
		printk("\n");

	{
		struct signal_queue *q;
		char s[sizeof(sigset_t)*2+1], b[sizeof(sigset_t)*2+1];

		render_sigset_t(&p->signal, s);
		render_sigset_t(&p->blocked, b);
		printk("   sig: %d %s %s :", signal_pending(p), s, b);
		for (q = p->sigqueue; q ; q = q->next)
			printk(" %d", q->info.si_signo);
		printk(" X\n");
	}
}

char * render_sigset_t(sigset_t *set, char *buffer)
{
	int i = _NSIG, x;
	do {
		i -= 4, x = 0;
		if (sigismember(set, i+1)) x |= 1;
		if (sigismember(set, i+2)) x |= 2;
		if (sigismember(set, i+3)) x |= 4;
		if (sigismember(set, i+4)) x |= 8;
		*buffer++ = (x < 10 ? '0' : 'a' - 10) + x;
	} while (i >= 4);
	*buffer = 0;
	return buffer;
}

void show_state(void)
{
	struct task_struct *p;

#if (BITS_PER_LONG == 32)
	printk("\n"
	       "                         free                        sibling\n");
	printk("  task             PC    stack   pid father child younger older\n");
#else
	printk("\n"
	       "                                 free                        sibling\n");
	printk("  task                 PC        stack   pid father child younger older\n");
#endif
	read_lock(&tasklist_lock);
	for_each_task(p)
		show_task(p);
	read_unlock(&tasklist_lock);
}

/*
 *	Put all the gunge required to become a kernel thread without
 *	attached user resources in one place where it belongs.
 */

void daemonize(void)
{
	struct fs_struct *fs;


	/*
	 * If we were started as result of loading a module, close all of the
	 * user space pages.  We don't need them, and if we didn't close them
	 * they would be locked into memory.
	 */
	exit_mm(current);
```

```c
        current->session = 1;
        current->pgrp = 1;

        /* Become as one with the init task */

        exit_fs(current);       /* current->fs->count--; */
        fs = init_task.fs;
        current->fs = fs;
        atomic_inc(&fs->count);

}

void __init init_idle(void)
{
        struct schedule_data * sched_data;
        sched_data = &aligned_data[smp_processor_id()].schedule_data;

        if (current != &init_task && task_on_runqueue(current)) {
                printk("UGH! (%d:%d) was on the runqueue, removing.\n",
                        smp_processor_id(), current->pid);
                del_from_runqueue(current);
        }
        sched_data->curr = current;
        sched_data->last_schedule = get_cycles();
}

extern void init_timervecs (void);

void __init sched_init(void)
{
        /*
         * We have to do a little magic to get the first
         * process right in SMP mode.
         */
        int cpu = smp_processor_id();
        int nr;

        init_task.processor = cpu;

        for(nr = 0; nr < PIDHASH_SZ; nr++)
                pidhash[nr] = NULL;

        init_timervecs();

        init_bh(TIMER_BH, timer_bh);
        init_bh(TQUEUE_BH, tqueue_bh);
        init_bh(IMMEDIATE_BH, immediate_bh);

        /*
         * The boot idle thread does lazy MMU switching as well:
         */
        atomic_inc(&init_mm.mm_count);
        enter_lazy_tlb(&init_mm, current, cpu);
}
```