Rusty’s Unreliable Guide To Kernel Hacking

Paul ‘Rusty’ Russell

IBM

Revision History
Revision 1 Tue Aug 27 2002

The Linux Kernel contains 5 million lines of source. It is difficult to know where to start, when you want to modify it in some way. This tutorial will cover various kernel programming conceps in depth: you will gain an appreciation for the Linux kernel by reading some of the code some of the great programmers, including of Linus Torvalds and Ingo Molnar, and an insight into Linux kernel development methods and politics (this is a preliminary version, the final version for the Congress may still suffer some changes. The Editors).

Table of Contents

1. Target Audience 2
2. Principles 2
    2.1. The Kernel Tree 2
    2.2. Overview 3
    2.3. Fundamental Constraints 3
    2.4. The Three Kernel Entry Points 4
3. Booting 5
4. System Calls & Traps 5
    4.1. System Calls & Traps 5
    4.2. Hardware Interrupts 6
    4.3. Software Interrupts 6
    4.4. Multiple CPU Reference Table 6
    4.5. Race Conditions 7
    4.6. Race Conditions 8
    4.7. Common Library Functions 10
5. Programming 10
6. Practicalities 14

6.1. Kernel Numbering 15
6.2. Where To Get Kernel Source 15
6.3. Editing the Kernel 15
6.4. GCC/ISO C Extensions 16
6.5. More Common Functions 17
7. Programming 17
8. Politics 40
8.1. linux-kernel Mailing List 40
8.2. Patches 41
8.3. Style 41
9. References 42
10. Appendix A: Ingo Molnar on Linux Accepting Patches 42
11. Appendix B: Linus Torvalds on Coding Style 48

1. Target Audience

Programmers without Linux Kernel experience.

2. Principles

2.1. The Kernel Tree

```
linux-2.5.3
     |------ Documentation
     |------ arch
     |       |------ ...
     |       |------ i386
     |       |       |------ boot
     |       |       |       |------ compressed
     |       |       |       `------ tools
     |       |       |------ kernel
     |       |       |       `------ cpu
     |       |       |------ lib
     |       |       |------ math-emu
     |       |       |------ mm
     |       |       `------ pci
     |       |------ ...
     |------ drivers
     |------ fs
     |------ include
```

```
|       |------ ...
|       |------ asm-i386
|       |------ ...
|       |------ linux
|       |------ ...
|------ init
|------ ipc
|------ kernel
|------ lib
|------ mm
|------ net
|------ scripts
|------ security
`------ sound
```

2.2. Overview

- 17 architectures
- 12,000 files, 5,000,000 lines of code

drivers	drivers/: 3,200 files, 2,300,000 lines of code
arch	arch/: 2,900 files, 990,000 lines of code
filesystems	fs/: 660 files, 330,000 lines of code
networking	net/: 450 files, 230,000 lines of code
core	init/ kernel/ lib/ mm/: 100 files, 46,000 lines of code

We will NOT be covering all these today!

2.3. Fundamental Constraints

Kernel is written in GNU C & inline assembler (arch specific code).

```
static inline int foo(void) { return -ENOMEM }
```

Global address space. No memory protection: you can destroy everything.

```
int *foo = NULL;
*foo = 1; /* BOOM! */
```

No libc in the kernel: only a small subset.

```
malloc(), strtoul()
```

No floating point or MMX usage.

```
int x = foo * 2.5; /* BOOM! */
```

Limited stack space (shared with interrupts on some archs)

```
char buffer[4096]; /* BOOM! */
```

Platform independence: 64-bit clean, endian-clean, char-signed clean

```
int foo = (int)&ptr; /* BOOM! */
char *firstbyte = (char &)&foo;  /* BOOM! */
if (*firstbyte < 0) /* BOOM! */
```

2.4. The Three Kernel Entry Points

1. Booting

2. System Calls

3. Interrupts

3. Booting

```
arch/<arch>/kernel/head.S


init/main.c


        calls module_init( ) and __setup( ) functions


        mounts root filesystem


        execs init process
```

4. System Calls & Traps

```
arch/<arch>/kernel/entry.S
        Hard-coded table of functions
```

Functions called from here must be declared asmlinkage.

Negative return sets errno, eg "return -ENOENT;"

Traps are similar to asynchronous system calls. eg. page fault, segmentation fault

4.1. System Calls & Traps

We call these "user context"

- Doing work for a particular process

- Process accessed using the "current" global variable

We can switch to other tasks voluntarily, by calling "schedule()". Sometimes called "sleeping".

We can be preempted by other tasks, too.

4.2. Hardware Interrupts

```
request_irq( ) registers your hardware interrupt handler


arch/<arch>/kernel/irq.c:do_IRQ( ) calls your handler.


Interrupts are disabled while your handler is running, so be quick!
```

4.3. Software Interrupts

Three types: softirqs, tasklets, bottom halves.

Marked (usually by hardware interrupt handlers) to do more work.

Timer functions (add_timer et al) run as bottom halves.

Run on return from hw handlers

```
... and from ksoftirqd if very busy.
```

See simultaneous.eps

4.4. Multiple CPU Reference Table

```
                        Soft Interrupts
User-  User-   Bottom Tasklet Softirq IRQ
space  context Half
```

Same one runs simultaneously on other CPU?	No	No	No	No	Yes	No
Same type runs simultaneously on other CPU?	Yes	Yes	No	Yes	Yes	Yes
Interrupted by same type?	Yes	Yes*	No	No	No	No
Interrupted by soft interrupts?	Yes	Yes	No	No	No	No
Interrupted by hard interrupts?	Yes	Yes	Yes	Yes	Yes	No

4.5. Race Conditions

Problems:

- Shared address space: all using the same global & static variables.
- Other functions can interrupt at any time
- Maybe multiple CPUs (SMP)

We need ways to protect data from simultaneous access.

We call these multiple accesses at the same time "Race Conditions"

foo.c:

```
/* Increment i by 1 */
i++;
```

foo.s:

```
lwz 9,0(3)      # Load contents of R3 + 0 into R9
addi 9,9,1      # Add one to R9
stw 9,0(3)      # Put contents of R9 back into R3 + 0
```

4.6. Race Conditions

Three flavours of race conditions in the Linux Kernel:

4.6.1. Race conditions when a processor is interrupted

```
lwz 9,0(3)      # Load contents of R3 + 0 into R9
        ***** INTERRUPT *****
        ...
        lwz 9,0(3)      # Load contents of R3 + 0 into R9
        addi 9,9,1      # Add one to R9
        stw 9,0(3)      # Put contents of R9 back into R3 + 0
        ...
        ***** RETURN FROM INTERRUPT *****
addi 9,9,1      # Add one to R9
stw 9,0(3)      # Put contents of R9 back into R3 + 0
```

4.6.2. Race conditions when a process is preempted

```
lwz 9,0(3)      # Load contents of R3 + 0 into R9
        ***** PROCESS 1 KICKED OFF CPU.  PROCESS 2: *****
        ...
        lwz 9,0(3)      # Load contents of R3 + 0 into R9
```

```
        addi 9,9,1      # Add one to R9
        stw 9,0(3)      # Put contents of R9 back into R3 + 0
        ...
        ***** PROCESS 1 RETURNS TO CPU *****
addi 9,9,1      # Add one to R9
stw 9,0(3)      # Put contents of R9 back into R3 + 0
```

4.6.3. Race conditions caused by multiple processors (SMP)

```
CPU 1                   CPU 2
...
lwz 9,0(3)              ...
addi 9,9,1              lwz 9,0(3)
stw 9,0(3)              addi 9,9,1
...                     stw 9,0(3)
                        ...
```

Protect from interruption by hardware interrupts:

```
local_irq_disable(int irq) & local_irq_enable(int irq)
```

Protection from software interrupts:

```
local_bh_disable(void) & local_bh_enable(void)
```

Protection from other CPUs:

```
spin_lock(spinlock_t *) & spin_unlock(spinlock_t *)
```

Preemption by other user contexts:

```
preempt_disable(void) & preempt_enable(void)
```

Combinations:

```
spin_lock_bh & spin_unlock_bh
spin_lock_irq & spin_unlock_irq
spin_lock_irqsave & spin_unlock_irqrestore
```

Reader-writer:

```
read_lock*/write_lock* & read_unlock*/write_unlock*
```

We cannot call any function which might sleep (schedule()) while using ANY of these.

There are special primitives for protecting from other user contexts while in user context: they sleep if they have to wait:

```
down_interruptible(struct semaphore *) & up(struct semaphore *)
```

```
down_read(struct rw_semaphore *) & up_read(struct rw_semaphore *)
down_write(struct rw_semaphore *) & up_write(struct rw_semaphore *)
```

4.7. Common Library Functions

```
kmalloc(size, flags) & kfree(ptr)
vmalloc(size) & vfree(ptr)
copy_from_user/get_user & copy_to_user/put_user
printk(format, ... )
udelay(int usecs) & mdelay(int msecs )
include/linux/list.h
```

5. Programming

`kernel/softirq.c`

```
/*
```

```
 *      linux/kernel/softirq.c
 *
 *      Copyright (C) 1992 Linus Torvalds
 *
 * Fixed a disable_bh()/enable_bh() race (was causing a console lockup)
 * due bh_mask_count not atomic handling. Copyright (C) 1998  Andrea Arcangeli
 *
 * Rewritten. Old one was good in 2.2, but in 2.3 it was immoral. --ANK (990903)

 */



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



/*
   - No shared variables, all the data are CPU local.
   - If a softirq needs serialization, let it serialize itself
     by its own spinlocks.
   - Even if softirq is serialized, only local cpu is marked for
     execution. Hence, we get something sort of weak cpu binding.
     Though it is still not clear, will it result in better locality
     or will not.



   Examples:
   - NET RX softirq. It is multithreaded and does not require
     any global serialization.
   - NET TX softirq. It kicks software netdevice queues, hence
     it is logically serialized per device, but this serialization
     is invisible to common code.
   - Tasklets: serialized wrt itself.
   - Bottom halves: globally serialized, grr...
 */
```

```
irq_cpustat_t irq_stat[NR_CPUS];


static struct softirq_action softirq_vec[32] __cacheline_aligned_in_smp;


/*
 * we cannot loop indefinitely here to avoid userspace starvation,
 * but we also don't want to introduce a worst case 1/HZ latency
 * to the pending events, so lets the scheduler to balance
 * the softirq load for us.
 */
static inline void wakeup_softirqd(unsigned cpu)
{
        struct task_struct * tsk = ksoftirqd_task(cpu);


        if (tsk && tsk->state != TASK_RUNNING)
                wake_up_process(tsk);
}


asmlinkage void do_softirq()
{
        __u32 pending;
        long flags;
        __u32 mask;
        int cpu;


        if (in_interrupt())
                return;


        local_irq_save(flags);
        cpu = smp_processor_id();
```

```
        pending = softirq_pending(cpu);



        if (pending) {
                struct softirq_action *h;



                mask = ~pending;
                local_bh_disable();
restart:
                /* Reset the pending bitmask before enabling irqs */
                softirq_pending(cpu) = 0;



                local_irq_enable();



                h = softirq_vec;



                do {
                        if (pending & 1)
                                h->action(h);
                        h++;
                        pending >>= 1;
                } while (pending);



                local_irq_disable();



                pending = softirq_pending(cpu);
                if (pending & mask) {
                        mask &= ~pending;
                        goto restart;
                }
                __local_bh_enable();
```

```
                if (pending)
                        wakeup_softirqd(cpu);
        }


        local_irq_restore(flags);
}


/*
 * This function must run with irqs disabled!
 */
inline void cpu_raise_softirq(unsigned int cpu, unsigned int nr)
{
        __cpu_raise_softirq(cpu, nr);


        /*
         * If we're in an interrupt or bh, we're done
         * (this also catches bh-disabled code). We will
         * actually run the softirq once we return from
         * the irq or bh.
         *
         * Otherwise we wake up ksoftirqd to make sure we
         * schedule the softirq soon.
         */
        if (!in_interrupt())
                wakeup_softirqd(cpu);
}


void raise_softirq(unsigned int nr)
{
        long flags;


        local_irq_save(flags);
        cpu_raise_softirq(smp_processor_id(), nr);
        local_irq_restore(flags);
}
```

6. Practicalities

6.1. Kernel Numbering

Even middle point is stable (eg. 2.4.17).

- Bug fixes
- New drivers

Odd is unstable (eg. 2.5.3).

- Everything can change

Unstable becomes stable every few years.

-pre or -rc kernels are designed for testing only.

Currently playing: 2.5. Linus et. al.

Stable: 2.4. Marcelo et al.

6.2. Where To Get Kernel Source

Kernels come from ftp.es.kernel.org/pub/linux/kernel/

Download patches and apply them by hand:

```
cp -al linux-2.5.30 linux-2.5.31
cd linux-2.5.31
zcat ../patch-2.5.31.gz | patch -p1
```

6.3. Editing the Kernel

Recommended method (if editor breaks hardlinks!):

```
cd ~/kernel-sources
wget http://www.moses.uklinux.net/patches/dontdiff
```

```
grab-kernel 2.5.3 .
cp -al linux-2.5.3 working-2.5.3-myhacks
cd working-2.5.3-myhacks
```

To produce a patch:

```
cd ~/kernel-sources
diff -urN -X ~/dontdiff linux-2.5.3 working-2.5.3-myhacks | grep -v Binary
```

Make sure you read the patch before you send it out!

More hints can be found: http://www.kernelnewbies.org

6.4. GCC/ISO C Extensions

Named structure initializers

```
struct foo bar = {
        .func = myfunc,
};
```

inline functions

```
static inline int myfunc(void)
{
        return -ENOSYS;
}
```

Variable argument macros

```
#define DEBUG(x,...) printk(KERN_DEBUG x , __VA_ARGS__)
```

Statement expressions ({ and })

```
#define get_cpu() ({ preempt_disable(); smp_processor_id(); })
```

__builtin_constant_p()

```
#define test_bit(nr,addr)                       (__builtin_constant_p(nr) ?
```

6.5. More Common Functions

```
wait_queue_head_t: DECLARE_WAIT_QUEUE_HEAD( )/init_waitqueue_head( )
wait_event_interruptible(wq, condition)
wake_up(wait_queue_head_t *)
```

```
HZ & jiffies
add_timer( ) & del_timer_sync( )
```

```
struct completion: DECLARE_COMPLETION( )/init_completion( )
wait_for_completion(struct completion *)
complete(struct completion *)
```

7. Programming

kernel/sched.c

```
/*
 *  kernel/sched.c
 *
 *  Kernel scheduler and related syscalls
 *
 *  Copyright (C) 1991-2002  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
 *  2002-01-04  New ultra-scalable O(1) scheduler by Ingo Molnar:
```

```
 *              hybrid priority-list and round-robin design with
 *              an array-switch method of distributing timeslices
 *              and per-CPU runqueues.  Cleanups and useful suggestions
 *              by Davide Libenzi, preemptible kernel bits by Robert Love.
 */


#include <linux/mm.h>
#include <linux/nmi.h>
#include <linux/init.h>
#include <asm/uaccess.h>
#include <linux/highmem.h>
#include <linux/smp_lock.h>
#include <asm/mmu_context.h>
#include <linux/interrupt.h>
#include <linux/completion.h>
#include <linux/kernel_stat.h>
#include <linux/security.h>
#include <linux/notifier.h>
#include <linux/delay.h>



/*
 * Convert user-nice values [ -20 ... 0 ... 19 ]
 * to static priority [ MAX_RT_PRIO..MAX_PRIO-1 ],
 * and back.
 */
#define NICE_TO_PRIO(nice)      (MAX_RT_PRIO + (nice) + 20)
#define PRIO_TO_NICE(prio)      ((prio) - MAX_RT_PRIO - 20)
#define TASK_NICE(p)            PRIO_TO_NICE((p)->static_prio)



/*
 * 'User priority' is the nice value converted to something we
 * can work with better when scaling various scheduler parameters,
 * it's a [ 0 ... 39 ] range.
 */
#define USER_PRIO(p)            ((p)-MAX_RT_PRIO)
#define TASK_USER_PRIO(p)       USER_PRIO((p)->static_prio)
#define MAX_USER_PRIO           (USER_PRIO(MAX_PRIO))



/*
```

```
 * These are the ’tuning knobs’ of the scheduler:
 *
 * Minimum timeslice is 10 msecs, default timeslice is 150 msecs,
 * maximum timeslice is 300 msecs. Timeslices get refilled after
 * they expire.
 */
#define MIN_TIMESLICE           ( 10 * HZ / 1000)
#define MAX_TIMESLICE           (300 * HZ / 1000)
#define CHILD_PENALTY           95
#define PARENT_PENALTY          100
#define EXIT_WEIGHT             3
#define PRIO_BONUS_RATIO        25
#define INTERACTIVE_DELTA       2
#define MAX_SLEEP_AVG           (2*HZ)
#define STARVATION_LIMIT        (2*HZ)



/*
 * If a task is ’interactive’ then we reinsert it in the active
 * array after it has expired its current timeslice. (it will not
 * continue to run immediately, it will still roundrobin with
 * other interactive tasks.)
 *
 * This part scales the interactivity limit depending on niceness.
 *
 * We scale it linearly, offset by the INTERACTIVE_DELTA delta.
 * Here are a few examples of different nice levels:
 *
 *  TASK_INTERACTIVE(-20): [1,1,1,1,1,1,1,1,1,0,0]
 *  TASK_INTERACTIVE(-10): [1,1,1,1,1,1,1,0,0,0,0]
 *  TASK_INTERACTIVE(  0): [1,1,1,1,0,0,0,0,0,0,0]
 *  TASK_INTERACTIVE( 10): [1,1,0,0,0,0,0,0,0,0,0]
 *  TASK_INTERACTIVE( 19): [0,0,0,0,0,0,0,0,0,0,0]
 *
 * (the X axis represents the possible -5 ... 0 ... +5 dynamic
 *  priority range a task can explore, a value of ’1’ means the
 *  task is rated interactive.)
 *
 * Ie. nice +19 tasks can never get ’interactive’ enough to be
 * reinserted into the active array. And only heavily CPU-hog nice -20
 * tasks will be expired. Default nice 0 tasks are somewhere between,
 * it takes some effort for them to get interactive, but it’s not
 * too hard.
 */
```

```
#define SCALE(v1,v1_max,v2_max)                  (v1) * (v2_max) / (v1_max)


#define DELTA(p)                  (SCALE(TASK_NICE(p), 40, MAX_USER_PRIO*PRIO_BONUS_RATIO/100


#define TASK_INTERACTIVE(p)                  ((p)->prio <= (p)->static_prio - DELTA(p))



/*
 * BASE_TIMESLICE scales user-nice values [ -20 ... 19 ]
 * to time slice values.
 *
 * The higher a thread's priority, the bigger timeslices
 * it gets during one round of execution. But even the lowest
 * priority thread gets MIN_TIMESLICE worth of execution time.
 *
 * task_timeslice() is the interface that is used by the scheduler.
 */


#define BASE_TIMESLICE(p) (MIN_TIMESLICE +                  ((MAX_TIMESLICE - MIN_TIMESLICE)



static inline unsigned int task_timeslice(task_t *p)
{
        return BASE_TIMESLICE(p);
}


/*
 * These are the runqueue data structures:
 */
```

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


typedef struct runqueue runqueue_t;


struct prio_array {
        int nr_active;
        unsigned long bitmap[BITMAP_SIZE];
        list_t queue[MAX_PRIO];
};


/*
 * This is the main, per-CPU runqueue data structure.
 *
 * Locking rule: those places that want to lock multiple runqueues
 * (such as the load balancing or the thread migration code), lock
 * acquire operations must be ordered by ascending &runqueue.
 */
struct runqueue {
        spinlock_t lock;
        unsigned long nr_running, nr_switches, expired_timestamp,
                        nr_uninterruptible;
        task_t *curr, *idle;
        prio_array_t *active, *expired, arrays[2];
        int prev_nr_running[NR_CPUS];


        task_t *migration_thread;
        list_t migration_queue;


} ____cacheline_aligned;


static struct runqueue runqueues[NR_CPUS] __cacheline_aligned;
```

```
#define cpu_rq(cpu)             (runqueues + (cpu))
#define this_rq()               cpu_rq(smp_processor_id())
#define task_rq(p)              cpu_rq(task_cpu(p))
#define cpu_curr(cpu)           (cpu_rq(cpu)->curr)
#define rt_task(p)              ((p)->prio < MAX_RT_PRIO)



/*
 * Default context-switch locking:
 */
#ifndef prepare_arch_switch
# define prepare_arch_switch(rq, next)  do { } while(0)
# define finish_arch_switch(rq, next)   spin_unlock_irq(&(rq)->lock)
# define task_running(rq, p)            ((rq)->curr == (p))
#endif



/*
 * task_rq_lock - lock the runqueue a given task resides on and disable
 * interrupts.  Note the ordering: we can safely lookup the task_rq without
 * explicitly disabling preemption.
 */
static inline runqueue_t *task_rq_lock(task_t *p, unsigned long *flags)
{
        struct runqueue *rq;



repeat_lock_task:
        local_irq_save(*flags);
        rq = task_rq(p);
        spin_lock(&rq->lock);
        if (unlikely(rq != task_rq(p))) {
                spin_unlock_irqrestore(&rq->lock, *flags);
                goto repeat_lock_task;
        }
        return rq;
}



static inline void task_rq_unlock(runqueue_t *rq, unsigned long *flags)
{
        spin_unlock_irqrestore(&rq->lock, *flags);
```

```
}



/*
 * rq_lock - lock a given runqueue and disable interrupts.
 */
static inline runqueue_t *this_rq_lock(void)
{
        runqueue_t *rq;



        local_irq_disable();
        rq = this_rq();
        spin_lock(&rq->lock);



        return rq;
}



static inline void rq_unlock(runqueue_t *rq)
{
        spin_unlock(&rq->lock);
        local_irq_enable();
}



/*
 * Adding/removing a task to/from a priority array:
 */
static inline void dequeue_task(struct task_struct *p, prio_array_t *array)
{
        array->nr_active--;
        list_del(&p->run_list);
        if (list_empty(array->queue + p->prio))
                __clear_bit(p->prio, array->bitmap);
}



static inline void enqueue_task(struct task_struct *p, prio_array_t *array)
{
```

```
        list_add_tail(&p->run_list, array->queue + p->prio);
        __set_bit(p->prio, array->bitmap);
        array->nr_active++;
        p->array = array;
}



/*
 * effective_prio - return the priority that is based on the static
 * priority but is modified by bonuses/penalties.
 *
 * We scale the actual sleep average [0 .... MAX_SLEEP_AVG]
 * into the -5 ... 0 ... +5 bonus/penalty range.
 *
 * We use 25% of the full 0...39 priority range so that:
 *
 * 1) nice +19 interactive tasks do not preempt nice 0 CPU hogs.
 * 2) nice -20 CPU hogs do not get preempted by nice 0 tasks.
 *
 * Both properties are important to certain workloads.
 */
static inline int effective_prio(task_t *p)
{
        int bonus, prio;



        bonus = MAX_USER_PRIO*PRIO_BONUS_RATIO*p->sleep_avg/MAX_SLEEP_AVG/100 -

                        MAX_USER_PRIO*PRIO_BONUS_RATIO/100/2;



        prio = p->static_prio - bonus;
        if (prio < MAX_RT_PRIO)
                prio = MAX_RT_PRIO;
        if (prio > MAX_PRIO-1)
                prio = MAX_PRIO-1;
        return prio;
}



/*
 * activate_task - move a task to the runqueue.
```

```
 * Also update all the scheduling statistics stuff. (sleep average
 * calculation, priority modifiers, etc.)
 */
static inline void activate_task(task_t *p, runqueue_t *rq)
{
        unsigned long sleep_time = jiffies - p->sleep_timestamp;
        prio_array_t *array = rq->active;



        if (!rt_task(p) && sleep_time) {
                /*
                 * This code gives a bonus to interactive tasks. We update
                 * an 'average sleep time' value here, based on
                 * sleep_timestamp. The more time a task spends sleeping,
                 * the higher the average gets - and the higher the priority
                 * boost gets as well.
                 */
                p->sleep_avg += sleep_time;
                if (p->sleep_avg > MAX_SLEEP_AVG)
                        p->sleep_avg = MAX_SLEEP_AVG;
                p->prio = effective_prio(p);
        }
        enqueue_task(p, array);
        rq->nr_running++;
}



/*
 * deactivate_task - remove a task from the runqueue.
 */
static inline void deactivate_task(struct task_struct *p, runqueue_t *rq)
{
        rq->nr_running--;
        if (p->state == TASK_UNINTERRUPTIBLE)
                rq->nr_uninterruptible++;
        dequeue_task(p, p->array);
        p->array = NULL;
}



/*
 * resched_task - mark a task 'to be rescheduled now'.
 *
```

```
 * On UP this means the setting of the need_resched flag, on SMP it
 * might also involve a cross-CPU call to trigger the scheduler on
 * the target CPU.
 */
static inline void resched_task(task_t *p)
{
#ifdef CONFIG_SMP
        int need_resched, nrpolling;



        preempt_disable();
        /* minimise the chance of sending an interrupt to poll_idle() */
        nrpolling = test_tsk_thread_flag(p,TIF_POLLING_NRFLAG);
        need_resched = test_and_set_tsk_thread_flag(p,TIF_NEED_RESCHED);
        nrpolling |= test_tsk_thread_flag(p,TIF_POLLING_NRFLAG);



        if (!need_resched && !nrpolling && (task_cpu(p) != smp_processor_id()))

                smp_send_reschedule(task_cpu(p));
        preempt_enable();
#else
        set_tsk_need_resched(p);
#endif
}



/***
 * try_to_wake_up - wake up a thread
 * @p: the to-be-woken-up thread
 * @sync: do a synchronous wakeup?
 *
 * Put it on the run-queue if it's not already there. The "current"
 * thread 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.
 *
 * returns failure only if the task is already active.
 */
static int try_to_wake_up(task_t * p, int sync)
{
        unsigned long flags;
        int success = 0;
```

```
        long old_state;
        runqueue_t *rq;


repeat_lock_task:
        rq = task_rq_lock(p, &flags);
        old_state = p->state;
        if (!p->array) {
                /*
                 * Fast-migrate the task if it's not running or runnable
                 * currently. Do not violate hard affinity.
                 */
                if (unlikely(sync && !task_running(rq, p) &&
                        (task_cpu(p) != smp_processor_id()) &&
                        (p->cpus_allowed & (1UL << smp_processor_id())))) {


                        set_task_cpu(p, smp_processor_id());
                        task_rq_unlock(rq, &flags);
                        goto repeat_lock_task;
                }
                if (old_state == TASK_UNINTERRUPTIBLE)
                        rq->nr_uninterruptible--;
                activate_task(p, rq);


                if (p->prio < rq->curr->prio)
                        resched_task(rq->curr);
                success = 1;
        }
        p->state = TASK_RUNNING;
        task_rq_unlock(rq, &flags);


        return success;
}


int wake_up_process(task_t * p)
{
        return try_to_wake_up(p, 0);
```

```
}


/*
 * context_switch - switch to the new MM and the new
 * thread's register state.
 */
static inline task_t * context_switch(task_t *prev, task_t *next)
{
        struct mm_struct *mm = next->mm;
        struct mm_struct *oldmm = prev->active_mm;



        if (unlikely(!mm)) {
                next->active_mm = oldmm;
                atomic_inc(&oldmm->mm_count);
                enter_lazy_tlb(oldmm, next, smp_processor_id());
        } else
                switch_mm(oldmm, mm, next, smp_processor_id());



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



        /* Here we just switch the register state and the stack. */
        switch_to(prev, next, prev);



        return prev;
}



/*
 * double_rq_lock - safely lock two runqueues
 *
 * Note this does not disable interrupts like task_rq_lock,
 * you need to do so manually before calling.
 */
```

```
static inline void double_rq_lock(runqueue_t *rq1, runqueue_t *rq2)
{
        if (rq1 == rq2)
                spin_lock(&rq1->lock);
        else {
                if (rq1 < rq2) {
                        spin_lock(&rq1->lock);
                        spin_lock(&rq2->lock);
                } else {
                        spin_lock(&rq2->lock);
                        spin_lock(&rq1->lock);
                }
        }
}



/*
 * double_rq_unlock - safely unlock two runqueues
 *
 * Note this does not restore interrupts like task_rq_unlock,
 * you need to do so manually after calling.
 */
static inline void double_rq_unlock(runqueue_t *rq1, runqueue_t *rq2)
{
        spin_unlock(&rq1->lock);
        if (rq1 != rq2)
                spin_unlock(&rq2->lock);
}



#if CONFIG_SMP



/*
 * double_lock_balance - lock the busiest runqueue
 *
 * this_rq is locked already. Recalculate nr_running if we have to
 * drop the runqueue lock.
 */
static inline unsigned int double_lock_balance(runqueue_t *this_rq,
        runqueue_t *busiest, int this_cpu, int idle, unsigned int nr_running)
{
        if (unlikely(!spin_trylock(&busiest->lock))) {
                if (busiest < this_rq) {
```

```
                        spin_unlock(&this_rq->lock);
                        spin_lock(&busiest->lock);
                        spin_lock(&this_rq->lock);
                        /* Need to recalculate nr_running */
                        if (idle || (this_rq->nr_running > this_rq->prev_nr_running[this_cpu]

                                nr_running = this_rq->nr_running;
                        else
                                nr_running = this_rq->prev_nr_running[this_cpu];

                } else
                        spin_lock(&busiest->lock);
        }
        return nr_running;
}



/*
 * find_busiest_queue - find the busiest runqueue.
 */
static inline runqueue_t *find_busiest_queue(runqueue_t *this_rq, int this_cpu, int idle, int

{
        int nr_running, load, max_load, i;
        runqueue_t *busiest, *rq_src;



        /*
         * We search all runqueues to find the most busy one.
         * We do this lockless to reduce cache-bouncing overhead,
         * we re-check the 'best' source CPU later on again, with
         * the lock held.
         *
         * We fend off statistical fluctuations in runqueue lengths by
         * saving the runqueue length during the previous load-balancing
         * operation and using the smaller one the current and saved lengths.
         * If a runqueue is long enough for a longer amount of time then
         * we recognize it and pull tasks from it.
         *
         * The 'current runqueue length' is a statistical maximum variable,
         * for that one we take the longer one - to avoid fluctuations in
         * the other direction. So for a load-balance to happen it needs
         * stable long runqueue on the target CPU and stable short runqueue
         * on the local runqueue.
         *
```

```
 * We make an exception if this CPU is about to become idle - in
 * that case we are less picky about moving a task across CPUs and
 * take what can be taken.
 */
if (idle || (this_rq->nr_running > this_rq->prev_nr_running[this_cpu]))

        nr_running = this_rq->nr_running;
else
        nr_running = this_rq->prev_nr_running[this_cpu];



busiest = NULL;
max_load = 1;
for (i = 0; i < NR_CPUS; i++) {
        if (!cpu_online(i))
                continue;



        rq_src = cpu_rq(i);
        if (idle || (rq_src->nr_running < this_rq->prev_nr_running[i]))

                load = rq_src->nr_running;
        else
                load = this_rq->prev_nr_running[i];
        this_rq->prev_nr_running[i] = rq_src->nr_running;



        if ((load > max_load) && (rq_src != this_rq)) {
                busiest = rq_src;
                max_load = load;
        }
}



if (likely(!busiest))
        goto out;



*imbalance = (max_load - nr_running) / 2;
```

```
        /* It needs an at least ~25% imbalance to trigger balancing. */
        if (!idle && (*imbalance < (max_load + 3)/4)) {
                busiest = NULL;
                goto out;
        }



        nr_running = double_lock_balance(this_rq, busiest, this_cpu, idle, nr_running);

        /*
         * Make sure nothing changed since we checked the
         * runqueue length.
         */
        if (busiest->nr_running <= nr_running + 1) {
                spin_unlock(&busiest->lock);
                busiest = NULL;
        }
out:
        return busiest;
}



/*
 * pull_task - move a task from a remote runqueue to the local runqueue.
 * Both runqueues must be locked.
 */
static inline void pull_task(runqueue_t *src_rq, prio_array_t *src_array, task_t *p, runqueue

{
        dequeue_task(p, src_array);
        src_rq->nr_running--;
        set_task_cpu(p, this_cpu);
        this_rq->nr_running++;
        enqueue_task(p, this_rq->active);
        /*
         * Note that idle threads have a prio of MAX_PRIO, for this test
         * to be always true for them.
         */
        if (p->prio < this_rq->curr->prio)
                set_need_resched();
}
```

```
/*
 * Current runqueue is empty, or rebalance tick: if there is an
 * inbalance (current runqueue is too short) then pull from
 * busiest runqueue(s).
 *
 * We call this with the current runqueue locked,
 * irqs disabled.
 */
static void load_balance(runqueue_t *this_rq, int idle)
{
        int imbalance, idx, this_cpu = smp_processor_id();
        runqueue_t *busiest;
        prio_array_t *array;
        list_t *head, *curr;
        task_t *tmp;



        busiest = find_busiest_queue(this_rq, this_cpu, idle, &imbalance);
        if (!busiest)
                goto out;



        /*
         * We first consider expired tasks. Those will likely not be
         * executed in the near future, and they are most likely to
         * be cache-cold, thus switching CPUs has the least effect
         * on them.
         */
        if (busiest->expired->nr_active)
                array = busiest->expired;
        else
                array = busiest->active;



new_array:
        /* Start searching at priority 0: */
        idx = 0;
skip_bitmap:
        if (!idx)
                idx = sched_find_first_bit(array->bitmap);
        else
                idx = find_next_bit(array->bitmap, MAX_PRIO, idx);
        if (idx == MAX_PRIO) {
```

```
                if (array == busiest->expired) {
                        array = busiest->active;
                        goto new_array;
                }
                goto out_unlock;
        }



        head = array->queue + idx;
        curr = head->prev;
skip_queue:
        tmp = list_entry(curr, task_t, run_list);



        /*
         * We do not migrate tasks that are:
         * 1) running (obviously), or
         * 2) cannot be migrated to this CPU due to cpus_allowed, or
         * 3) are cache-hot on their current CPU.
         */



#define CAN_MIGRATE_TASK(p,rq,this_cpu)                                         ((j



        curr = curr->prev;



        if (!CAN_MIGRATE_TASK(tmp, busiest, this_cpu)) {
                if (curr != head)
                        goto skip_queue;
                idx++;
                goto skip_bitmap;
        }
        pull_task(busiest, array, tmp, this_rq, this_cpu);
        if (!idle && --imbalance) {
                if (curr != head)
                        goto skip_queue;
                idx++;
                goto skip_bitmap;
        }
```

```
out_unlock:
        spin_unlock(&busiest->lock);
out:
        ;
}



/*
 * One of the idle_cpu_tick() and busy_cpu_tick() functions will
 * get called every timer tick, on every CPU. Our balancing action
 * frequency and balancing agressivity depends on whether the CPU is
 * idle or not.
 *
 * busy-rebalance every 250 msecs. idle-rebalance every 1 msec. (or on
 * systems with HZ=100, every 10 msecs.)
 */
#define BUSY_REBALANCE_TICK (HZ/4 ?: 1)
#define IDLE_REBALANCE_TICK (HZ/1000 ?: 1)



static inline void idle_tick(runqueue_t *rq)
{
        if (jiffies % IDLE_REBALANCE_TICK)
                return;
        spin_lock(&rq->lock);
        load_balance(rq, 1);
        spin_unlock(&rq->lock);
}



#endif



/*
 * We place interactive tasks back into the active array, if possible.
 *
 * To guarantee that this does not starve expired tasks we ignore the
 * interactivity of a task if the first expired task had to wait more
 * than a ’reasonable’ amount of time. This deadline timeout is
 * load-dependent, as the frequency of array switched decreases with
 * increasing number of running tasks:
 */
#define EXPIRED_STARVING(rq)                        ((rq)->expired_timestamp &&
```

```
/*
 * This function gets called by the timer code, with HZ frequency.
 * We call it with interrupts disabled.
 */
void scheduler_tick(int user_ticks, int sys_ticks)
{
        int cpu = smp_processor_id();
        runqueue_t *rq = this_rq();
        task_t *p = current;



        if (p == rq->idle) {
                /* note: this timer irq context must be accounted for as well */

                if (irq_count() - HARDIRQ_OFFSET >= SOFTIRQ_OFFSET)
                        kstat.per_cpu_system[cpu] += sys_ticks;
#if CONFIG_SMP
                idle_tick(rq);
#endif
                return;
        }
        if (TASK_NICE(p) > 0)
                kstat.per_cpu_nice[cpu] += user_ticks;
        else
                kstat.per_cpu_user[cpu] += user_ticks;
        kstat.per_cpu_system[cpu] += sys_ticks;



        /* Task might have expired already, but not scheduled off yet */
        if (p->array != rq->active) {
                set_tsk_need_resched(p);
                return;
        }
        spin_lock(&rq->lock);
        if (unlikely(rt_task(p))) {
                /*
                 * RR tasks need a special form of timeslice management.
                 * FIFO tasks have no timeslices.
                 */
                if ((p->policy == SCHED_RR) && !--p->time_slice) {
                        p->time_slice = task_timeslice(p);
```

```
                        p->first_time_slice = 0;
                        set_tsk_need_resched(p);



                        /* put it at the end of the queue: */
                        dequeue_task(p, rq->active);
                        enqueue_task(p, rq->active);
                }
                goto out;
        }
        /*
         * The task was running during this tick - update the
         * time slice counter and the sleep average. Note: we
         * do not update a thread's priority until it either
         * goes to sleep or uses up its timeslice. This makes
         * it possible for interactive tasks to use up their
         * timeslices at their highest priority levels.
         */
        if (p->sleep_avg)
                p->sleep_avg--;
        if (!--p->time_slice) {
                dequeue_task(p, rq->active);
                set_tsk_need_resched(p);
                p->prio = effective_prio(p);
                p->time_slice = task_timeslice(p);
                p->first_time_slice = 0;



                if (!TASK_INTERACTIVE(p) || EXPIRED_STARVING(rq)) {
                        if (!rq->expired_timestamp)
                                rq->expired_timestamp = jiffies;
                        enqueue_task(p, rq->expired);
                } else
                        enqueue_task(p, rq->active);
        }
out:
#if CONFIG_SMP
        if (!(jiffies % BUSY_REBALANCE_TICK))
                load_balance(rq, 0);
#endif
        spin_unlock(&rq->lock);
}
```

```
void scheduling_functions_start_here(void) { }



/*
 * schedule() is the main scheduler function.
 */
asmlinkage void schedule(void)
{
        task_t *prev, *next;
        runqueue_t *rq;
        prio_array_t *array;
        list_t *queue;
        int idx;



        if (unlikely(in_interrupt()))
                BUG();



#if CONFIG_DEBUG_HIGHMEM
        check_highmem_ptes();
#endif
need_resched:
        preempt_disable();
        prev = current;
        rq = this_rq();



        release_kernel_lock(prev);
        prev->sleep_timestamp = jiffies;
        spin_lock_irq(&rq->lock);



        /*
         * if entering off of a kernel preemption go straight
         * to picking the next task.
         */
        if (unlikely(preempt_count() & PREEMPT_ACTIVE))
                goto pick_next_task;
```

```
        switch (prev->state) {
        case TASK_INTERRUPTIBLE:
                if (unlikely(signal_pending(prev))) {
                        prev->state = TASK_RUNNING;
                        break;
                }
        default:
                deactivate_task(prev, rq);
        case TASK_RUNNING:
                ;
        }
pick_next_task:
        if (unlikely(!rq->nr_running)) {
#if CONFIG_SMP
                load_balance(rq, 1);
                if (rq->nr_running)
                        goto pick_next_task;
#endif
                next = rq->idle;
                rq->expired_timestamp = 0;
                goto switch_tasks;
        }



        array = rq->active;
        if (unlikely(!array->nr_active)) {
                /*
                 * Switch the active and expired arrays.
                 */
                rq->active = rq->expired;
                rq->expired = array;
                array = rq->active;
                rq->expired_timestamp = 0;
        }



        idx = sched_find_first_bit(array->bitmap);
        queue = array->queue + idx;
        next = list_entry(queue->next, task_t, run_list);



switch_tasks:
        prefetch(next);
```

```
        clear_tsk_need_resched(prev);



        if (likely(prev != next)) {
                rq->nr_switches++;
                rq->curr = next;



                prepare_arch_switch(rq, next);
                prev = context_switch(prev, next);
                barrier();
                rq = this_rq();
                finish_arch_switch(rq, prev);
        } else
                spin_unlock_irq(&rq->lock);



        reacquire_kernel_lock(current);
        preempt_enable_no_resched();
        if (test_thread_flag(TIF_NEED_RESCHED))
                goto need_resched;
}
```

8. Politics

Linus is God

"The Linux kernel has no core team, and we all know who they are"

Alan Cox (paraphrased)

8.1. linux-kernel Mailing List

You do not need to know anything to post to Linux Kernel:

```
From: Rusty Russell <rusty@linuxcare.com.au>
To: root@chaos.analogic.com
Cc: lkml <linux-kernel@vger.kernel.org>
Subject: Re: On labelled initialisers, gcc-2.7.2.3 and tcp_ipv4.c
```

```
Date: Mon, 16 Oct 2000 20:59:38 +1100


In message <Pine.LNX.3.95.1001015203310@chaos.analogic.com> you write:
> The 'C' language can order structure members anyway it wants.


You are an idiot.


Rusty.
--
Hacking time.
```

Discussion on linux-kernel is usually best when code is included

8.2. Patches

Patches get dropped

- Even good patches.
- Trivial Patch Monkey: trivial@rustcorp.com.au

Small number of people have Linus bandwidth reservation

8.3. Style

Documentation/CodingStyle

```
        #ifdefs are bad, try to avoid them in functions.
                eg. Define dummy functions for non-SMP.
```

Linus says: ALWAYS _RETURN_ THE ERROR.

Usually aim for the *smallest possible patch*: If Linus says "please do more", then great!

Revolutionary changes have been most successful when they are small (at least to begin with)

The correct way to write a portable driver is to write it for 2.5, and then use compatibility macros for 2.4 and 2.2.

Removal of debugging stuff: if it's only useful to the maintainer, get rid of it before submission.

Standards are much higher for core code and infrastructure then drivers and other code which doesn't break anything else.

Do not optimize your locking: use locking idioms.

Do not use typedefs unneccessarily.

9. References

1. Linux kernel sources:

    • Documentation/CodingStyle

    • Documentation/DocBook/kernel-hacking.sgml

    • Documentation/DocBook/kernel-locking.sgml

2. Unix Systems for Modern Architectures:

    • Symmetric Multiprocessing and Caching for Kernel Programmers

    • Curt Schimmel [ISBN: 0201633388]

3. Linux Device Drivers

4. Understanding the Linux Kernel

10. Appendix A: Ingo Molnar on Linux Accepting Patches

```
Date:   Tue, 29 Jan 2002 14:54:27 +0100 (CET)
From: Ingo Molnar <mingo@elte.hu>
Reply-To: <mingo@elte.hu>
To: Rob Landley <landley@trommello.org>
Cc: Linus Torvalds <torvalds@transmeta.com>,
       <linux-kernel@vger.kernel.org>
Subject: Re: A modest proposal -- We need a patch penguin
In-Reply-To: <200201290446.g0T4kZU31923@snark.thyrsus.com>
Message-ID: <Pine.LNX.4.33.0201291324560.3610-100000@localhost.localdomain>
MIME-Version: 1.0
Content-Type: TEXT/PLAIN; charset=US-ASCII
Sender: linux-kernel-owner@vger.kernel.org
Precedence: bulk
X-Mailing-List: linux-kernel@vger.kernel.org



On Mon, 28 Jan 2002, Rob Landley wrote:



> (You keep complaining people never send you patches.  People are
> suggesting automated patch remailers to spam your mailbox even harder.
> There has GOT to be a better way...)



None of the examples you cited so far are convincing to me, and i'd like
to explain why. I've created and submitted thousands of patches to the
Linux kernel over the past 4 years (my patch archive doesnt go back more
than 4 years):



 # ls patches | wc -l
    2818



a fair percentage of those went to Linus as well, and while having seen
some of them rejected does hurt mentally, i couldnt list one reject from
Linus that i wouldnt reject *today*. But i sure remember being frustrated
about rejects when they happened. In any case, i have some experience in
submitting patches and i'm maintaining a few subsystems, so here's my take
```

on the ’patch penguin’ issue:

If a patch gets ignored 33 times in a row then perhaps the person doing the patch should first think really hard about the following 4 issues:

- cleanliness
- concept
- timing
- testing

a violation of any of these items can cause patch to be dropped *without notice*. Face it, it’s not Linus’ task to teach people how to code or how to write correct patches. Sure, he still does teach people most of the time, but you cannot *expect* him to be able to do it 100% of the time.

1) cleanliness

code cleanliness is a well-know issue, see Documentation/CodingStyle. If a patch has such problems then maintainers are very likely to help - Linus probably wont and shouldnt. I’m truly shocked sometimes, how many active and experienced kernel developers do not follow these guidelines. While the Linux coding style might be arbitrary in places, all coding styles are arbitrary in some areas, and only one thing is important above all: consistency between kernel subsystems. If i go from one kernel subsystem to another then i’d like to have the same ’look and feel’ of source code - i think this is a natural desire we all share. If anyone doesnt see the importance of this issue then i’d say he hasnt seen, hacked and maintained enough kernel code yet. I’d say the absolute positive example here is Al Viro. I think most people just do not realize the huge amount of background cleanup work Al did in the past 2 years. And guess what? I bet Linus would be willing to apply Al’s next patch blindfolded.

impact: a patch penguin might help here - but he probably wont scale as well as the current set of experienced kernel hackers scale, many of whom

are happy to review patches for code cleanliness (and other) issues.

2) concept

many of the patches which were rejected for a long time are *difficult* issues. And one thing many patch submitters miss: even if the concept of the patch is correct, you first have to start by cleaning up *old* code, see issue 1). Your patch is not worth a dime if you leave in old cruft, or if the combination of old cruft and your new code is confusing. Also, make sure the patch is discussed and developed openly, not on some obscure list. linux-kernel@vger.kernel.org will do most of the time. I do not want to name specific patches that violate this point (doing that in public just offends people needlessly - and i could just as well list some of my older patches), but i could list 5 popular patches immediately.

impact: a patch penguin just wont solve this concept issue, because, by definition, he doesnt deal with design issues. And most of the big patch rejections happen due to exactly these concept issues.

3) timing

kernel source code just cannot go through arbitrary transitions. Eg. right now the scheduler is being cleaned up (so far it was more than 50 sub-patches and they are still coming) - and work is going on to maximize the quality of the preemption patch, but until the base scheduler has stabilized there is just no point in applying the preemption patch - no matter how good the preemption patch is. Robert understands this very much. Many other people do not.

impact: a patch penguin just wont solve this issue, because a patch penguin cannot let his tree transition arbitrarily either. Only separately maintained and tested patches/trees can handle this issue.

4) testing

there are code areas and methods which need more rigorous testing and third-party feedback - no matter how good the patch. Most notably, if a patch exports some new user-space visible interface, then this item applies. An example is the aio patch, which had all 3 items right but was rejected due to this item. [things are improving very well on the aio front so i think this will change in the near future.]

impact: a patch penguin just wont solve this issue, because his job, by definition, is not to keep patches around indefinitely, but to filter them to Linus. Only separately maintained patches/trees help here. More people are willing to maintain separate trees is good (-dj, -ac, -aa, etc.), one tree can do a nontrivial transition at a time, and by having more of them we can eg. get one of them testing aio, the other one testing some other big change. A single patch penguin will be able to do only one nontrivial transition - and it's not his task to do nontrivial transitions to begin with.

Many people who dont actually maintain any Linux code are quoting Rik's complains as an example. I'll now have to go on record disagreeing with Rik humbly, i believe he has done a number of patch-management mistakes during his earlier VM development, and i strongly believe the reason why Linus ignored some of his patches were due to these issues. Rik's flames against Linus are understandable but are just that: flames. Fortunately Rik has learned meanwhile (we all do) and his rmap patches are IMHO top-notch. Joining the Andrea improvements and Rik's tree could provide a truly fantastic VM. [i'm not going to say anything about the IDE patches situation because while i believe Rik understands public criticism, i failed to have an impact on Andre before :-)]

also, many people just start off with a single big patch. That just doesnt work and you'll likely violate one of the 4 items without even noticing it. Start small, because for small patches people will have the few minutes needed to teach you. The bigger a patch, the harder it is to review it, and the less likely it happens. Also, if a few or your patches have gone into the Linux tree that does not mean you are senior kernel hacker and can start off writing the one big, multi-megabyte super-feature

you dreamt about for years. Start small and increase the complexity of your patches slowly - and perhaps later on you'll notice that that super-feature isnt all that super anymore. People also underestimate the kind of complexity explosion that occurs if a large patch is created. Instead of 1-2 places, you can create 100-200 problems.

face it, most of the patches rejected by Linus are not due to overload. He doesnt guarantee to say why he rejects patches - *and he must not*. Just knowing that your patch got rejected and thinking it all over again often helps finding problems that Linus missed first time around. If you submit to Linus then you better know exactly what you do.

if you are uncertain about why a patch got rejected, then shake off your frustration and ask *others*. Many kernel developers, including myself, are happy to help reviewing patches. But people do have egos, and it happens very rarely that people ask it on public lists why their patches got rejected, because people do not like talking about failures. And the human nature makes it much easier to attack than to talk about failures. Which fact alone pretty much shows that most of the time the problem is with the patch submitter, not with Linus.

it's so much easier to blame Linus, or maintainers. It's so much easier to fire off an email flaming Linus and getting off the steam than to actually accept the possibility of mistake and *fix* the patch. I'll go on record saying that good patches are not ignored, even these days when the number of active kernel hackers has multipled. People might have to go through several layers first, and finding some kernel hacker who is not as loaded as Linus to review your patch might be necessery as well (especially if the patch is complex), but if you go through the right layers then you can be sure that nothing worthwile gets rejected arbitrarily.

Ingo

-
To unsubscribe from this list: send the line "unsubscribe linux-kernel" in the body of a message to majordomo@vger.kernel.org
More majordomo info at http://vger.kernel.org/majordomo-info.html

```
Please read the FAQ at  http://www.tux.org/lkml/
```

11. Appendix B: Linus Torvalds on Coding Style

```
Date:   Wed, 30 Aug 2000 10:04:12 -0700 (PDT)
From: Linus Torvalds <torvalds@transmeta.com>
To: Rogier Wolff <R.E.Wolff@BitWizard.nl>
Cc: Arnaldo Carvalho de Melo <acme@conectiva.com.br>,
        Philipp Rumpf <prumpf@parcelfarce.linux.theplanet.co.uk>,
        Kenneth Johansson <ken@canit.se>, Jean-Paul Roubelat <jpr@f6fbb.org>,
        davem@redhat.com, linux-kernel@vger.kernel.org
Subject: Re: [PATCH] af_rose.c: s/suser/capable/ + micro cleanups
In-Reply-To: <200008300717.JAA02899@cave.bitwizard.nl>
Message-ID: <Pine.LNX.4.10.10008300941230.1393-100000@penguin.transmeta.com>
MIME-Version: 1.0
Content-Type: TEXT/PLAIN; charset=US-ASCII
Sender: linux-kernel-owner@vger.kernel.org
Precedence: bulk
X-Mailing-List:         linux-kernel@vger.kernel.org



On Wed, 30 Aug 2000, Rogier Wolff wrote:
>
> > source code smaller and more easier to read (yes, this is debatable,
> > I think it becomes more clean, other think otherwise, I'm just
> > following what Linus said he prefer).
>
> The kernel is a multi-million-lines-of-code piece of software.
> Software maintenance cost is found to correlate strongly with the
> number of lines-of-code.
>
> So, I would prefer the shorter version.



I disagree.



Number of lines is irrelevant.
```

The _complexity_ of lines counts.

And ?: is a complex construct, that is not always visually very easy to parse because of the "non-local" behaviour.

That is not saying that I think you shouldn’t use ?: at all. It’s a wonderful construct in many ways, and I use it all the time myself. But I actually prefer

```
        if (complex_test)
                return complex_expression1;

        return complex_expression2;
```

over

```
        return (complex_test) ? complex_expression1 : complex_expression2;
```

because by the time you have a complex ?: thing it’s just not very readable any more.

Basically, dense lines are bad. And ?: can make for code that ends up "too dense".

More specific example: I think

```
        return copy_to_user(dst, src, size) ? -EFAULT : 0;
```

is fine and quite readable. Fits on a simple line.

However, it’s getting iffy when it becomes something like

```
        return copy_to_user(buf, page_address(page) + offset, size) ? -EFAULT: 0;
```

for example. The "return" is so far removed from the actual return values, that it takes some parsing (maybe you don’t even see everything on an 80-column screen, or even worse, you split up one expression over several lines..

(Basically, I don’t like multi-line expressions. Avoid stuff like

```
        x = ...
                + ...
                - ...;
```

unless it is _really_ simple. Similarly, some people split up their "for ()" or "while ()" statement things - which usually is just a sign of the loop baing badly designed in the first place. Multi-line expressions are sometimes unavoidable, but even then it’s better to try to simplify them as much as possible. You can do it by many means

- make an inline function that has a descriptive name. It’s still complex, but now the complexity is _described_, and not mixed in with potentially other complex actions.

- Use logical grouping. This is sometimes required especially in "if()" statements with multiple parts (ie "if ((x || y) && !z)" can easily become long - but you might consider just the above inline function or #define thing).

- Use multiple statements. I personally find it much more readable to have

```
        if (PageTestandClearReferenced(page))
                goto dispose_continue;

        if (!page->buffers && page_count(page) > 1)
                goto dispose_continue;

        if (TryLockPage(page))
                goto dispose_continue;
```

  rather than the equivalent

```
        if (PageTestandClearReferenced(page) ||
            (!page->buffers && page_count(page) > 1) ||
            TryLockPage(page))
                goto dispose_continue;
```

  regardless of any grouping issues.

Basically, lines-of-code is a completely bogus metric for _anything_.
Including maintainability.

> If it takes you a few seconds to look this over, that’s fine. Even it
> the one "complicated" line take twice as long (per line) as the
> original 4 lines, then it’s a win.

I disagree violently.

Linus