# Real Time Optimizations for the Linux Kernel and the PREEMPT_RT patch

Chris Winsor

CS-502 Term Project

10-January-2012

# Table of Contents

# Introduction

This paper reviews Real-time optimizations for the Linux kernel.

- CONFIG_PREEMPT_VOLUNTARY
- CONFIG_PREEMPT
- PREEMPT_RT Patch

The first two are configuration switches. The third is a patch.

# How to get the code

CONFIG_PREEMPT_VOLUNTARY and CONFIG_PREEMPT are configuration switches which are part of the baseline kernel source. As such they are available directly from www.kernel.org and also from downstream distributions. They require only a recompile of the kernel.

PREEMPT_RT is a patch. It is available directly from www.kernel.org for version 2.6 up to version 2.6.33 and also for versions 3.0 and 3.2. Downstream sources may (or may not) distribute it depending on the priority of real time for their target audience and the resources at their disposal. For example OpenSUSE provided support up to about 2 years ago then dropped it due to lack of resources. Ubuntu (with a sizable robotics audience) currently distributes it.

# Terminology and Context

Before diving into code it is necessary to define terminology and give some context.

## *Interrupt Latency*

Interrupt Latency is defined as the time from an interrupt being issued to the time when the bottom part of the interrupt service routine starts (1).

## *Servicing an Interrupt*

The process of servicing an interrupt involves the following steps:

a) Operating system will receive the interrupt from the hardware (execute the top-half interrupt service routine)

b) Operating system will pre-empt the current task

c) Operating will switch context and begin execution of the bottom-half interrupt service routine

## *Preemption in the Linux Kernel*

Preemption (step b in interrupt servicing) is the primary source of uncontrolled latency for the Linux kernel. Preemption in the vanilla (non-patched) Linux kernel can occur at the following times:

- upon return from an interrupt handler to user-context code (user preemption)
- upon return from kernel-context code to user-context code (user preemption)
- upon return from an interrupt handler to kernel-context code (kernel preemption)
- when kernel-context code releases its last lock and becomes preemptible (kernel preemption)
- if a task in the kernel explicitly calls schedule () (volunteers to be pre-empted) (kernel preemption)
- if a task in the kernel blocks (which results in a call to schedule()) (kernel preemption)

As such there are three cases where kernel-mode thread can NOT be preempted. These are:

- A kernel stream waiting for a lock to be acquired.
- A kernel stream running in interrupt context
- A kernel stream that is cpu bound (does not release a lock, volunteer or block)

We will see that these are the cases that the CONFIG_PREEMPT and PREEMPR_RT patch go after.

# Code Analysis

Source and patch are from kernel.org, version 2.6.33.9.

## *CONFIG_PREEMPT_VOLUNTARY (configuration switch)*

CONFIG_PREEMPT_VOLUNTARY is the first configuration switch for the Linux kernel. A kernel built with this switch set will have checks to sources of long latencies in the kernel code. A kernel task can voluntarily allow a higher priority task to pre-empt it.

### The Code:

*In [kernel.h]*

```
#ifdef CONFIG_PREEMPT_VOLUNTARY
# define might_resched() _cond_resched()
#else
# define might_resched() do { } while (0)
#endif
```

*Summary: under CONFIG_PREEMPT_VOLUNTARY might_resched() which formally was stubbed out, becomes active.*

*In [sched.c]*

```
int __sched _cond_resched(void)
{
        if (should_resched()) {
                __cond_resched();
                return 1;
        }
        return 0;
}
```

*Summary: _cond_resched uses should_resched() to see if the scheduler should be called. If the task has volunteered to be rescheduled should_resched will reflect this.*

### Bottom Line

*CONFIG_PREEMPT_VOLUNTARY provides a path to cond_resched if the task has volunteered to be rescheduled.*

## *CONFIG_PREEMPT (configuration switch)*

CONFIG_PREEMPT is the second configuration switch for the Linux kernel. This switch causes all kernel code outside of spinlock and interrupt handlers to be preemptible by a higher priority kernel thread.

## The Code:

*Spinlock, mutex, softirq and other functions have provisions for preemption points. These are indicated by the preempt_enable() function call.*

```
./spinlock.c:       preempt_enable();
./mutex.c:          preempt_enable();
./softirq.c:        preempt_enable();
./signal.c:  preempt_enable();
./smp.c:     preempt_enable();
./perf_event.c:     preempt_enable();
./kprobes.c:        preempt_enable();
./module.c:  preempt_enable();
./srcu.c:    preempt_enable();
```

*preempt_enable is defined n [preempt.h]*

```
#ifdef CONFIG_PREEMPT

#define preempt_disable() \
       inc_preempt_count(); \

#define preempt_enable_no_resched() \
       dec_preempt_count(); \

#define preempt_check_resched() \
       if (unlikely(test_thread_flag(TIF_NEED_RESCHED))) \
              preempt_schedule(); \

#define preempt_enable() \
       preempt_enable_no_resched(); \
       barrier(); \
       preempt_check_resched(); \

#else
#define preempt_disable()         do { } while (0)
#define preempt_enable_no_resched()     do { } while (0)
#define preempt_enable()          do { } while (0)
#define preempt_check_resched()         do { } while (0)
#endif
```

*Summary: if CONFIG_PREEMPT is set, the functions to enable,disable and check preempt count are enabled. Non-CONFIG_PREEMPT stubs out these calls.*

*preempt_enable() performs a preempt_enable_no_resched() followed by preempt_check_resched().*

*The call to preempt_check_resched() will in turn will call prempt_schedule()*

*In [sched.c]*

```
#ifdef CONFIG_PREEMPT
asmlinkage void __sched preempt_schedule(void)
{
...
       do {
              add_preempt_count(PREEMPT_ACTIVE);
              schedule();
              sub_preempt_count(PREEMPT_ACTIVE);
       } while (need_resched());
}
```

*Summary - preempt_schedule() in turn calls schedule() which is the task scheduler. This will conditionally re-schedule (aka preempt) the current task.*

## Bottom Line

While the non-CONFIG_PREMPT code includes calls to preempt_enable() from spinlock, mutex and a number of sources, the functions themselves are stubbed out. With CONFIG_PREEMPT these calls become active and a call to preempt_enable is shown to result in a call to the scheduler (schedule()); It is in this manner that CONFIG_PREEMPT enables preemption within kernel tasks.

# PREEMPT_RT (patch)

PREEMPT_RT is a patch to the kernel. There are three elements to this patch. As described in (2) this converts Linux into a fully preemptible kernel by:

- Making in-kernel locking-primitives (using spinlocks) preemptible though reimplementation with rtmutexes.
- Converting interrupt handlers into preemptible kernel threads
- Implementing priority inheritance for in-kernel spinlocks and semaphores

The sections that follow investigate these three changes in detail.

## *Making in-kernel spinlocks preemptible*

Changes to convert spinlocks into preemptible mutexes are analyzed below.

### Motivation

Both spinlock and mutex are locks used to protect critical data. Where they differ is in how they handle requesters waiting for the lock. A spinlock causes the process to loop, burning CPU cycles until the lock frees. A Mutex cause the process state to be changed to "waiting" and n entry in an event queue is made, the event of which will cause the task to resume execution.

But there is another distinction - preemptibility. Specifically when a mutex is waiting to acquire a lock it can be pre-empted (in fact a process that has been wait-queued has already been pre-empted). A task that is in a spinlock is in a non-preemptible state.

As part of PREEMPT_RT kernel-code spinlocks are replaced with mutexes. There is some performance tradeoff here. The wait-queueing and state changes of a mutex introduces overhead as compared to a spinlock. But this is accepted in exchange for preempibility of the mutex. The tradeoff avoids latency of a spinlock, which is the primary objective in real-time system design.

### Code Changes

*In the patched version of [include/linux/spinlock.h]*

```
#ifdef PREEMPT_RT
#include <linux/rt_lock.h>
#define spin_lock(lock)    rt_spin_lock(lock)
```

*In [include/linux/rt_lock.h]*

```
extern void __lockfunc rt_spin_lock(spinlock_t *lock);
```

*In [kernel/rtmutex.c]*

```
void __lockfunc rt_spin_lock(spinlock_t *lock)
{
        rt_spin_lock_fastlock(&lock->lock, rt_spin_lock_slowlock);
        spin_acquire(&lock->dep_map, 0, 0, _RET_IP_);
}
EXPORT_SYMBOL(rt_spin_lock);
```

*Summary: with the PREEMPT_RT patch, any code which calls "spin_lock()" now calls rt_spin_lock() instead.*

*rt_spin_lock() is defined in [rtmutex.c] (note "mutex") and this calls rt_spin_lock_fastlock() followed by spin_acquire(). As defined in [include/linux/lockdep.h] spin_acquire is stubbed out in non-debug modes.*

*We pursue rt_spin_lock_fastlock()...*

*in [kernel/rtmutex.c]*

```
static inline void rt_spin_lock_fastlock(struct rt_mutex *lock,
                void  (*slowfn)(struct rt_mutex *lock))
{
        if (likely(rt_mutex_cmpxchg(lock, NULL, current)))
                rt_mutex_deadlock_account_lock(lock, current);
        else
                slowfn(lock);
```

*Summary: a rt_mutex_cmpxchg is done on the lock. This does all the work (the subsequent rt_mutex_deadlock_account_lock is purely debug). rt_mutex_cmpxchg uses an architecture-dependent "cmpxchg" function...*

```
#if defined(__HAVE_ARCH_CMPXCHG) && !defined(CONFIG_DEBUG_RT_MUTEXES)
# define rt_mutex_cmpxchg(l,c,n)        (cmpxchg(&l->owner, c, n) == c)
#else
# define rt_mutex_cmpxchg(l,c,n)        (0)
```

*Summary: if the architecture does not support cmpxchg then rt_mutex_cmpxchg returns "0" and a "slowfn(lock)" is used instead.*

## Bottom line

PREEMPT_RT replaces "spin_lock" with a mutex "rt_mutex_cmpxchg". The advantage is the elimination of latency associated with kernel tasks in spinlock. There is a penalty of tasks being placed in the wait state for short periods and this is acceptable in exchange for the latency benefit in this case.

## *Converting interrupt handlers into preemptible kernel threads*

The second area of focus for PREEMPT_RT is the conversion of bottom-half interrupt processing to kernel threads thus making them preemptible.

### Terminology

The following terminology relating to interrupts are used to provide consistency in this section:

- **Interrupt processing** is the general term used to describe the servicing of interrupts.
- **Top half**, also referred to as the **interrupt handler** or **hard interrupt**, performs the initial part of interrupt processing, which is the task of receiving an interrupt from the hardware. This involves acknowledge receipt of the interrupt to hardware and notifying the bottom-half of the interrupt. The work performed by top-half is minimalistic; the intent is to push the majority of data processing activities to the bottom half.
- **Bottom half** is a term describing the latter portion of interrupt processing. In Linux kernel 2.5 and later bottom half interrupt handling is done via Softirq, Tasklet and Work Queue.
- **Softirq** is a mechanism to perform bottom-half interrupt processing. The infrastructure around Softirq consists of a softirq_pending mask, a statically allocated list of function pointers to bottom half routines, and a softirq handler. The softirq handler works by checking the softirq_pending mask and, if set, calling the function-pointer to the corresponding service routine. Softirq handler is called at the end of a top-half interrupt. There are nine softirq types two of which are dedicated to tasklets.
- **Tasklets** are layered on top of softirq. The primary element to support tasklets is a tasklet queue. Each entry in the queue identifies work to be performed, and as in softirq, this takes the form of a function pointer to bottom half interrupt processing routine. As compared to softirq, tasklets are a runtime mechanism. An entry to the tasklet queue can be made at any time, and each entry contains its own function pointer to a bottom-half routine.
- **Work queues** are a different mechanism to schedule bottom-half interrupt handling. In this case the work to be performed is represented by a task_struct, like a normal process.

## Context in Which Interrupt Code Runs

Recall that kernel code may be run in one of three contexts:

- interrupt context
- process context associated with a kernel thread
- process context associated with a user thread (kernel code running on behalf of user code)

In the vanilla (non-PREMPT_RT) code top-half, softirq and tasklets all run in interrupt context and are non-preemptible.

In PREMPT_RT the goal is to have bottom-half interrupt handling be preemptible. The effort then is to move softirq/tasklets into kernel process context. This change is summarized in the table below:

Kernel Context for Interrupt Processing in non-PREEMPT-RT and PREEMPT_RT

| | non-PREEMPT_RT | PREEMPT_RT |
|---|---|---|
| top-half | interrupt | interrupt |
| bottom-half softirq | interrupt | ***kernel thread*** |
| bottom-half tasklet | interrupt | ***kernel thread*** |
| bottom-half work queue | kernel thread | kernel thread |

PREEMPT_RT will run bottom–half softirq and tasklets in kernel thread context, whereas in non-PREEMPR_RT they were run in interrupt context.

## Justification

What justification is there for running existing bottom-half code in preemptible context where it was formerly in a non-preemptible context? Clearly there are large quantities of bottom-half interrupt processing code and these will not be rewritten as part of this process.

The justification comes from SMP (simultaneous multi-processor) support which has been part of the kernel from its earliest inception. SMP requires that kernel code be re-entrant - that is - more than one CPU can be executing it at the same time. This reentrant characteristic also applies to multiple threads executing the code at the same time. In other words - if the code is SMP safe it is also thread-safe. The bottom-half interrupt routines are SMP-safe (can be run by more than one CPU in parallel) so they can also be run by more than one software thread at the same time.

## Code (non-PREEMPT_RT)

First we look at non-PREEMPT_RT code.

*In [softirq.c]*

```
static struct softirq_action softirq_vec[NR_SOFTIRQS] __cacheline_aligned_in_smp;
static DEFINE_PER_CPU(struct task_struct *, ksoftirqd);
```

*Summary:*

*At compile time softirq_vec and ksoftirqd are created per CPU.*

*softirq_vec is the list of function pointers to bottom-half interrupt processing routines.*

*ksoftirq is a task_struct for a single kernel thread. This thread performs a clean-up role on soft interrupts when the system gets backlogged on softirqs. More specifically - when processing softirqs the system tries to process all softirqs and will loop on the softirq processing in an attempt to do this. When incoming softirq rate becomes high, confounded by the fact that some softirqs re-call themselves, time spent in this loop can be excessive. In this case the do_softirq will depart without all softirqs completed, leaving further softirq processing to the daemon.*

*[in softirq.c]*

```
void irq_exit(void)
{
        if (!in_interrupt() && local_softirq_pending())
                invoke_softirq();
        preempt_enable_no_resched();
}
# define invoke_softirq()        __do_softirq()
# define invoke_softirq()        do_softirq()
```

*Summary:*

*The activity of processing softirqs is initiated within irq_exit() upon departure from top-half interrupt handler. irq_exit calls invoke_softirq which in turn calls the main softirq processing function do_softirq()*

*[in softirq.c]*

```
#define MAX_SOFTIRQ_RESTART 10
asmlinkage void __do_softirq(void)
{
        struct softirq_action *h;
        __u32 pending;
        int max_restart = MAX_SOFTIRQ_RESTART;

        pending = local_softirq_pending();
```

```
        h = softirq_vec;

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


        pending = local_softirq_pending();
        if (pending && --max_restart)
                goto restart;

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

*Summary:*

*do_softirq is the primary mechanism to walk through softirqs and call the bottom half. pending is a mask indicating which softirqs require service. The highest priority softirq is bit 0. h is a function pointer to the bottom-half routine and is initialized to the first bottom-half routine softirq_vec[0]. The code iterates through the softirqs (highest priority first) and if service is needed performs h->action(h) to call the bottom half routine.*

*The routine tries to exit with a clear "pending" (all softirqs serviced) and will re-run itself (goto resetart) up to 10 times in this effort. But this may not be possible where interrupt rate is high or if softirqs frequently re-issue themselves. In this case the code decides to not starve the system and punts by waking the softirqd (kernel background thread) to continue the softirq processing.*

### *Bottom Line (non-PREEMPT_RT)*

To summarize the non-PREEMPT_RT code: the initial call to do_softirq comes from irq_exit() which is from the top-half interrupt and in the interrupt context. It is this (non-preemptible) interrupt context that PREEMPRT_RT desires to replace with preemptible kernel thread context.

## Code (PREEMPT_RT)

Now we look at PREEMPT_RT patched code.

*At compile time there are some changes to the data structures:*

```
static struct softirq_action softirq_vec[NR_SOFTIRQS] __cacheline_aligned_in_smp;
static DEFINE_PER_CPU(u32, softirq_running);
static DEFINE_PER_CPU(struct softirqdata [NR_SOFTIRQS], ksoftirqd);
```

*Summary:*

*softirq_vec is identical to that of non-RT code - it is still the list of bottom-half function call pointers.*

*A new structure "softirq_running" is defined. This is [PER_CPU].*

*ksoftirqd (the list of kernel threads) has been expanded from [PER_CPU] to be [PER_CPU][PER_SOFTIRQ]. There is to be a kernel thread for each softirq type.*

*Also the structure of ksoftirqd is now softirqdata, where it was formerly just task_struct. The new structure includes the original task_struct.*

```
struct softirqdata {
        int                      nr;
        unsigned long            cpu;
        struct task_struct      *tsk;
        int                      running;
};
```

*in [softirq.c] irq_exit() is unchanged - it still calls invoke_softirq()*

```
void irq_exit(void)
{
        if (!in_interrupt() && local_softirq_pending())
                invoke_softirq();
        __preempt_enable_no_resched();
}
# define invoke_softirq()        __do_softirq()
# define invoke_softirq()       do_softirq()
```

*Next we look for the function __do_softirq and find... the code formerly named "__do_softirq()" is now named "___do_softirq()" (3 undebars where there were 2 before!!). The new __do_softirq is as follows:*

```
in [softirq.c]
asmlinkage void __do_softirq(void)
{
#ifdef CONFIG_PREEMPT_SOFTIRQS
        /*
         * 'preempt harder'. Push all softirq processing off to ksoftirqd.
         */
        if (softirq_preemption) {
                if (local_softirq_pending())
                        trigger_softirqs();
                return;
        }
#endif
```

*Summary: __do_softirq function has been hijacked. Where formerly it started processing softirqs directly, it now calls trigger_softirqs.*

```
in [softirq.c]
/*
 * Wake up the softirq threads which have work
 */
```

```
static void trigger_softirqs(void)
{
        u32 pending = local_softirq_pending();
        int curr = 0;

        while (pending) {
                if (pending & 1)
                        wakeup_softirqd(curr);
                pending >>= 1;
                curr++;
        }
}
```

*Summary: trigger_softirqs() performs a wakup of softirqd kernel threads which handle softirqs.*

*So irq_exit does not directly initiate a softirq serving, rather wakes up a kernel thread to do this.*

*Lets see how this kernel thread does its job. To start with - the softirq kernel threads are created at the beginning of time via a callback called as part of system startup.*

*in [softirq.c]*

```
static int __cpuinit cpu_callback(struct notifier_block *nfb,
                                  unsigned long action,
                                  void *hcpu)
{
        int hotcpu = (unsigned long)hcpu, i;
        struct task_struct *p;

        switch (action) {
        case CPU_UP_PREPARE:
        case CPU_UP_PREPARE_FROZEN:
                for (i = 0; i < NR_SOFTIRQS; i++) {
                        per_cpu(ksoftirqd, hotcpu)[i].nr = i;
                        per_cpu(ksoftirqd, hotcpu)[i].cpu = hotcpu;
                        per_cpu(ksoftirqd, hotcpu)[i].tsk = NULL;
                }
                for (i = 0; i < NR_SOFTIRQS; i++) {
                        p = kthread_create(run_ksoftirqd,
                                           &per_cpu(ksoftirqd, hotcpu)[i],
                                           "sirq-%s/%d", softirq_names[i],
                                           hotcpu);
                        if (IS_ERR(p)) {
                                printk("ksoftirqd %d for %i failed\n", i,
                                       hotcpu);
                                return NOTIFY_BAD;
                        }
                        kthread_bind(p, hotcpu);
                        per_cpu(ksoftirqd, hotcpu)[i].tsk = p;
                }
                break;
        break;
```

*Summary:*

*As part of bringing up the system the [PER_CPU][PER_SOFTIRQ] kernel threads for processing softirqs are created. Prior to actually starting them the related ksoftirqd data structures are initialized. Then the*

*kernel threads are created (call to kthread_create). The starting point for the thread is the "run_ksoftirqd" function.*

*Looking at run_ksoftirqd [in softirq.c]*

```
static int run_ksoftirqd(void * __data)
{
        struct softirqdata *data = __data;
        u32 softirq_mask = (1 << data->nr);

        while (!kthread_should_stop()) {
                data->running = 1;
                while (local_softirq_pending() & softirq_mask) {
                        per_cpu(softirq_running, cpu) |= softirq_mask;
                        h = &softirq_vec[data->nr];
                        if (h)
                                h->action(h);
                        per_cpu(softirq_running, cpu) &= ~softirq_mask;
                }
                data->running = 0;
        }
        __set_current_state(TASK_RUNNING);
        return 0;
wait_to_die:
}
```

Summary:

*This code is similar to the original do_softirq. Here again, the code calls the softirq function via*

*h->action(h)*

*and there is a loop (local_softirq_pending() & softirq_mask) which is executed until the softirqs pending are clear. This time however we are running as a task and can be preempted.*

## *Bottom Line*

PREEMPT_RT patch introduces kernel threads for each softirq. These threads run in kernel context to process softirqs in a manner very similar to the original interrupt-context code. The primary difference is that the kernel thread provides preemptibility. If a higher priority task presents itself the bottom-half interrupt processing will be preempted.

## *Priority Inheritance for in-kernel spinlocks and semaphores*

Priority Inversion is defined as a sequence whereby a low priority task acquires a resource (such as a lock), then a medium priority task which is compute-bound preempts the low priority task. As a result a high priority task would be prevented access to the resource which is locked by the low priority task. The high priority task is blocked by a low priority task.

The solution taken by PREEMPT_RT is Priority Inheritance. Priority inheritance consists of bumping the priority of the low priority task that owns a lock when/if a higher priority task gets queued waiting for that same lock.

## The Code

To implement priority inheritance it is necessary to know the priorities of the current owner of the lock and that of the list of 'waiters' for that lock.

*in [rtmutex.h]*

*The rt_mutex structure is defined. An owner of the mutex (task_struct) is identified along with a list of 'waiters'.*

```
/**
 * The rt_mutex structure
 *
 * @wait_lock:      spinlock to protect the structure
 * @wait_list:      pilist head to enqueue waiters in priority order
 * @owner:    the mutex owner
 */
struct rt_mutex {
       raw_spinlock_t             wait_lock;
       struct plist_head   wait_list;
       struct task_struct  *owner;
};
```

*Priority inversion occurs during the process of acquiring a lock, when a task is blocked by a lower-priority task holding the lock. The call sequence is initiated by a call to rt_spin_lock.*

*in [rtmutex.c]*

```
void __lockfunc rt_spin_lock(spinlock_t *lock)
{
       rt_spin_lock_fastlock(&lock->lock, rt_spin_lock_slowlock);
       spin_acquire(&lock->dep_map, 0, 0, _RET_IP_);
```

```
}
```

*Summary: rt_spin_lock calls rt_spin_lock_fastlock with two parameters. The first is the lock, the second is a function pointer to a "slow path" function (rt_spin_lock_slowlock).*

```
static inline void
rt_spin_lock_fastlock(struct rt_mutex *lock,
              void  (*slowfn)(struct rt_mutex *lock))
{
       if (likely(rt_mutex_cmpxchg(lock, NULL, current)))
              rt_mutex_deadlock_account_lock(lock, current);
       else
              slowfn(lock);
}
```

*Summary - rt_spin_lock_fastlock takes a pointer to the lock, and a function pointer to a "slowlock" function.*

*It first attempts the "fast" (uncontested) lock acquisition (rt_mutex_cmpxchg).*

*We are interested in priority inversion which is where the lock is contended. In this case the "slow path" is needed and the function pointer is called.*

*From earlier that function as rt_spin_lock_slowlock().*

```
static void  noinline __sched
rt_spin_lock_slowlock(struct rt_mutex *lock)
{
       ret = task_blocks_on_rt_mutex(lock, &waiter, current, 0, flags, 1);
       for (;;) {

              /* Try to acquire the lock again. */
              if (do_try_to_take_rt_mutex(lock, current, &waiter, STEAL_LATERAL))
                     break;
                  }
```

*It is at this point where we are in a position of potential priority inversion. The current task is requesting a lock, and another (potentially lower priority) task may be holding it. It may be necessary to 'bump' the priority of the lock holder depending on the relative priority of the holder and the waiter. This is achieved by the call to task_blocks_on_rt_mutex(). This function, we will see will perform the priority bump. The code then loop waiting for the lock to become available by calling do_try_to_take_rt_mutex().*

*The priority bump is handled in the task_blocks_on_rt_mutex code:*

```
/*
 * Prepare waiter and propagate pi chain
 * This must be called with lock->wait_lock held.
 */
static int task_blocks_on_rt_mutex(struct rt_mutex *lock,
                                   struct rt_mutex_waiter *waiter,
                                   struct task_struct *task,
                                   int detect_deadlock, unsigned long flags,
                                   int savestate)
```

```
{

       res = rt_mutex_adjust_prio_chain(owner, detect_deadlock, lock, waiter,
                                        task);
}
```

*Which calls rt_mutex_adjust_prio_chain()*

```
/*
 * Adjust the priority chain. Also used for deadlock detection.
 * Decreases task's usage by one - may thus free the task.
 * Returns 0 or -EDEADLK.
 */
static int rt_mutex_adjust_prio_chain(struct task_struct *task,
                                      int deadlock_detect,
                                      struct rt_mutex *orig_lock,
                                      struct rt_mutex_waiter *orig_waiter,
                                      struct task_struct *top_task)
{

       waiter = task->pi_blocked_on;

       /* Grab the next task */
       task = rt_mutex_owner(lock);

       if (waiter == rt_mutex_top_waiter(lock)) {
              /* Boost the owner */
              __rt_mutex_adjust_prio(task);
       } else if (top_waiter == waiter) {
              /* Deboost the owner */
              __rt_mutex_adjust_prio(task);
       }
```

*The essential element is the call to __rt_mutex_adjust_prio(task). The "task" is the current owner of the lock. The call to __rt_mutext_adjust_prio() is to adjust the priority of that task. The current owner of the lock can be boosted or unboosted based on the priority of the waiting task.*

*in [rtmutex.c]*

```
/*
 * Adjust the priority of a task, after its pi_waiters got modified.
 * This can be both boosting and unboosting. task->pi_lock must be held.
 */
static void __rt_mutex_adjust_prio(struct task_struct *task)
{
       int prio = rt_mutex_getprio(task);

       if (task->prio != prio)
              rt_mutex_setprio(task, prio);
}
```

*And the priority gets set in [sched.h]*

```
static inline void rt_mutex_setprio(struct task_struct *p, int prio)
{
       task_setprio(p, prio);
}
```

### *Bottom Line*

An rt_mutex structure is defined which identifies the current owner of the lock and the list of waiting tasks. A call to request the lock can result in an uncontested fast-path, or if the lock is currently held, the slow path. This slow (contested) path is where priority inversion is avoided. This is done by reviewing the priority of the waiting task and comparing to that of the current lock holder. If the lock holder has priority lower than the waiter then the holder's priority is temporarily bumped to that of the waiter.

# Scheduling Policies and Debug Features

Linux code provides real-time-oriented scheduling policies and debug switches.

## *Scheduling Policies (SCHED_FIFO and SCHED_RR)*

The vanilla Linux kernel provides Real-time oriented scheduling policies (SCHED_FIFO and SCHED_RR). Structure around the real-time and non-real-time threads is necessary for these scheduling policies to be effective.

## *CONFIG_PREEMPT_NOTIFIERS (de bug switch)*

A set of callbacks are provided for purposes of debugging preemption. These are enabled via the CONFIG_PREEMPT_NOTIFIERS configuration switch. The callbacks are sched_in and sched_out. Sched_in is called just prior to a task being rescheduled, and sched_out is called just after having been preempted. With the callback comes a notifier (struct preempt_notifier) for the task being scheduled or pre-empted respectively, along with the CPU involved.

The callback list takes the form of a linked list in the task_struct (similar to how Lab 4 added a message list to the task struct).

## *CONFIG_PREEMPT_TRACER (debug switch)*

The CONFIG_PREEMPT_TRACER configuration switch

a) creates a global "irqsoff_trace" structure to track how long interrupts are disabled

b) enables debug messaging around preempt count underflow/overflow

In [trace_irqsoff.c] a static (single global) trace array is defined. This global array has, for each cpu, a "trace_array_cpu" with a bunch of information relating to preemption including preempt_timestamp, rt_priority, critical_start, critical_end and saved_latency. These are defined in [trace.h]

Then the functions start_critical_timings() and stop_critical_timings() are definied. This is done in [irqflags.h,trace/trace_irqsoff.c]. These functions establish start and stop timestamps used to measure time spent in critical sections.

start_critical_timing() is to be called at the beginning of a critical section. This function captures preempt_timestamp, flags and other data and posts to the global irqsoff_trace structcture.

stop_critical_timing() is to be called at the end of a critical section. This function captures flags and data and posts these to irqsoff_trace structure. It then calls "check_critical_timing()"

check_critical_timing() is defined in [trace/trace_irqsoff.c]. This function has access to the starting timestamp. It captures a current timestamp, computes a delta between the two, and checks/reports on latency.

# Factors in the Design of a Realtime System

Kernel configuration switches and patches do not by themselves constitute a well designed real-time system.

A structured approach to design starts by identifying **system requirements**. For RT system design this would certainly include maximum latency. The "requirements first" approach avoids the assumption that "all lower latencies are better" mindset, focusing instead on identifying the sources of long or unpredictable latencies to meet the requirements of the system.

A hardware platform may contain inherent sources of latency to the kernel that are outside the control of kernel.org. One example is **power management BIOS**. This code is provided by the hardware vendor and used to control CPU temperature using fan speed and clock rate. Kernel calls into BIOS are uninterruptible and unpreemptible and can expose the kernel to significant latencies. It is necessary to identify, understand, tune, or eliminate factors such as this as part of the design or test of a real-time system.

Many factors that contribute to a well designed real-time system, the examples above are just a few.

# Utilities for Benchmarking and Tuning

rt.wiki.kernel.org has a number of utilities, benchmarks, and test cases. These include tests used in Clark Williams' analysis (cyclictest, signaltest, pi_stress, sched_latency, sched_football, and pi-tests).

Tuning tools referred in the Williams presentation include tuna, ftrace, oprofile, and systemtap.

## Benchmark Results

It is not a goal of this paper to perform benchmarking. But it is interesting to look at others' results.

Clark Williams (architect at Red Hat) compared the vanilla kernel to PREEMPT_RT kernel. He presented the results at the 2008 Linux Realtime Summit. The presentation is here: http://people.redhat.com/bche/presentations/realtime-linux-summit08.pdf

The results are dramatic - PREEMPT_RT significantly reduced the standard deviation (variability) in interrupt latency. That is – latency was much more consistent (predictable) in PREEMPT_RT as compared to that of the vanilla kernel.

## In Summary

With CONFIG_PREEMPT_VOLUNTARY, CONFIG_PREEMPT and PREEMPT_RT Patch the Linux kernel is transformed into a system with very respectable real-time features. This broadens its target market and makes it viable as a candidate for all but the most rigorous real-time applications.

# References


(1) https://rt.wiki.kernel.org/articles/c/o/n/PREEMPT_RT_Patch_79df.html

(2) https://rt.wiki.kernel.org/


# Related Links and Articles:

- http://lwn.net/Articles/146861/
- https://lkml.org/lkml/2008/8/1/412 (proposed, not implemented)
- https://lkml.org/lkml/2011/7/28/332
- http://lwn.net/Articles/146861/
- https://lwn.net/Articles/440064/ (April 2011)
- https://help.ubuntu.com/community/UbuntuStudio/RealTimeKernel
- https://features.opensuse.org/306385
- http://forums.opensuse.org/english/get-technical-help-here/install-boot-login/447521-cannot-boot-suse-11-3-realtime-kernel.html
- http://forums.opensuse.org/english/get-technical-help-here/multimedia/462212-realtime-low-latency-kernels-opensuse-11-4-a.html
- http://www.kernel.org/pub/linux/kernel/projects/rt/
- http://www.kernel.org/pub/linux/kernel/