# Internals of the RT Patch

Steven Rostedt
*Red Hat, Inc.*
srostedt@redhat.com
rostedt@goodmis.org

Darren V. Hart
*IBM Linux Technology Center*
dvhltc@us.ibm.com

## Abstract

Steven Rostedt (srostedt@redhat.com)

Over the past few years, Ingo Molnar and others have worked diligently to turn the Linux kernel into a viable Real-Time platform. This work is kept in a patch that is held on Ingo's page of the Red Hat web site [7] and is referred to in this document as **the RT patch**. As the RT patch is reaching maturity, and slowly slipping into the upstream kernel, this paper takes you into the depths of the RT patch and explains exactly what it is going on. It explains Priority Inheritance, the conversion of Interrupt Service Routines into threads, and transforming spin_locks into mutexes and why this all matters. This paper is directed toward kernel developers that may eventually need to understand Real-Time (RT) concepts and help them deal with design and development changes as the mainline kernel heads towards a full fledge Real-Time Operating System (RTOS). This paper will offer some advice to help them avoid pitfalls that may come as the mainline kernel comes closer to an actual RTOS.

The RT patch has not only been beneficial to those in the Real-Time industry, but many improvements to the mainline kernel have come out of the RT patch. Some of these improvements range from race conditions that were fixed to reimplementation of major infrastructures.[1] The cleaner the mainline kernel is, the easier it is to convert it to an RTOS. When a change is made to the RT patch that is also beneficial to the mainline kernel, those changes are sent as patches to be incorporated into mainline.

[1] such as hrtimers and generic IRQs

## 1 The Purpose of a Real-Time Operating System

The goal of a Real-Time Operating System is to create a predictable and deterministic environment. The primary purpose is not to increase the speed of the system, or lower the latency between an action and response, although both of these increase the quality of a Real-Time Operating System. The primary purpose is to eliminate "surprises." A Real-Time system gives control to the user such that they can develop a system in which they can calculate the actions of the system under any given load with deterministic results. Increasing performance and lowering latencies help in this regard, but they are only second to deterministic behavior. A common misconception is that an RTOS will improve throughput and overall performance. A quality RTOS still maintains good performance, but an RTOS will sacrifice throughput for predictability.

To illustrate this concept, let's take a look at a hypothetical algorithm that on a non Real-Time Operating System, can complete some calculation in 250 microseconds on average. An RTOS on the same machine may take 300 microseconds for that same calculation. The difference is that an RTOS can guarantee that the worst case time to complete the calculation is known in advanced, and the time to complete the calculation will not go above that limit.[2] The non-RTOS can not guarantee a maximum upper limit time to complete that algorithm. The non-RTOS may perform it in 250 microseconds 99.9% of the time, but 0.1% of the time, it might take 2 milliseconds to complete. This is totally unacceptable for an RTOS, and may result in system failure. For example, that calculation may determine if a device driver needs to activate some trigger that must be set within 340 microseconds or the machine will lock up. So we see that a non-RTOS may have a better average

[2] when performed by the highest priority thread.

performance than an RTOS, but an RTOS guarantees to meet its execution time deadlines.

The above demonstrates an upper bound requirement for completing a calculation. An RTOS must also implement the requirement of response time. For example, a system may have to react to an asynchronous event. The event may be caused by an external stimulus (hitting a big red button) or something that comes from inside the system (a timer interrupt). An RTOS can guarantee a maximum response time from the time the stimulant occurs to the time the reaction takes place.

### 1.1 Latencies

The time between an event is expected to occur and the time it actually does is called **latency**. The event may be an external stimulus that wants a response, or a thread that has just woken up and needs to be scheduled. The following is the different kinds and causes of latencies and these terms will be used later in this paper.

- `Interrupt Latency` — The time between an interrupt triggering and when it is actually serviced.
- `Wakeup Latency` — The time between the highest priority task being woken up and the time it actually starts to run. This also can be called `Scheduling Latency`.
- `Priority Inversion` — The time a high priority thread must wait for a resource owned by a lower priority thread.
- `Interrupt Inversion` — The time a high priority thread must wait for an interrupt to perform a task that is of lower priority.

Interrupt latency is the easiest to measure since it corresponds tightly to the time interrupts are disabled. Of course, there is also the time that it takes to make it to the actual service routine, but that is usually a constant value.[3] The duration between the waking of a high priority process and it actually running is also a latency. This sometimes includes interrupt latency since waking of a process is usually due to some external event.

[3]except with the RT kernel, see Section 2.

Priority inversion is not a latency but the effect of priority inversion causes latency. The amount of time a thread must wait on a lower priority thread is the latency due to priority inversion. Priority inversion can not be prevented, but an RTOS must prevent unbounded priority inversion. There are several methods to address unbounded priority inversion, and Section 6 explains the method used by the RT patch.

Interrupt inversion is a type of priority inversion where a thread waits on an interrupt handler servicing a lower priority task. What makes this unique, is that a thread is waiting on an interrupt context that can not be preempted, as opposed to a thread that can be preempted and scheduled out. Section 2 explains how threaded interrupts address this issue.

## 2 Threaded Interrupts

As mentioned in Section 1.1, one of the causes of latency involves interrupts servicing lower priority tasks. A high priority task should not be greatly affected by a low priority task, for example, doing heavy disk IO. With the normal interrupt handling in the mainline kernel, the servicing of devices like hard-drive interrupts can cause large latencies for all tasks. The RT patch uses threaded interrupt service routines to address this issue.

When a device driver requests an IRQ, a thread is created to service this interrupt line.[4] Only one thread can be created per interrupt line. Shared interrupts are still handled by a single thread. The thread basically performs the following:

```
while (!kthread_should_stop()) {
  set_current_state
           (TASK_INTERRUPTIBLE);
  do_hardirq(desc);
  cond_resched();
  schedule();
}
```

Here's the flow that occurs when an interrupt is triggered:

The architecture function `do_IRQ()`[5] calls one of the following chip handlers:

[4]See `kernel/irq/manage.c` `do_irqd`.

[5]See `arch/<arch>/kernel/irq.c`. (May be different in some architectures.)

- handle_simple_irq
- handle_level_irq
- handle_fasteoi_irq
- handle_edge_irq
- handle_percpu_irq

Each of these sets the IRQ descriptor's status flag IRQ_INPROGRESS, and then calls redirect_hardirq().

redirect_hardirq() checks if threaded interrupts are enabled, and if the current IRQ is threaded (the IRQ flag IRQ_NODELAY is not set) then the associated thread (do_irqd) is awaken. The interrupt line is masked and the interrupt exits. The cause of the interrupt has not been handled yet, but since the interrupt line has been masked, that interrupt will not trigger again. When the interrupt thread is scheduled, it will handle the interrupt, clear the IRQ_INPROGRESS status flag, and unmask the interrupt line.

The interrupt priority inversion latency time is only the time from the triggering of the interrupt, the masking of the interrupt line, the waking of the interrupt thread, and returning back to the interrupted code, which takes on a modern computer system a few microseconds. With the RT patch, a thread may be given a higher priority than a device handler interrupt thread, so when the device triggers an interrupt, the interrupt priority inversion latency is only the masking of the interrupt line and waking the interrupt thread that will handle that interrupt. Since the high priority thread may be of a higher priority than the interrupt thread, the high priority thread will not have to wait for the device handler that caused that interrupt.

### 2.1 Hard IRQs That Stay Hard

It is important to note that there are cases where an interrupt service routine is not converted into a thread. Most notable example of this is the timer interrupt. The timer interrupt is handled in true interrupt context, and is not serviced by a thread. This makes sense since the timer interrupt controls the triggering of time events, such as, the scheduling of most threads.

A device can also specify that its interrupt handler shall be a true interrupt by setting the interrupt descriptor flag IRQ_NODELAY. This will force the interrupt handler to run in interrupt context and not as a thread. Also note that an IRQ_NODELAY interrupt can not be shared with threaded interrupt handlers. The only time that IRQ_NODELAY should be used is if the handler does very little and does not grab any spin_locks. If the handler acquires spin_locks, it will crash the system in full CONFIG_PREEMPT_RT mode.[6]

It is recommended never to use the IRQ_NODELAY flag unless you fully understand the RT patch. The RT patch takes advantage of the fact that interrupt handlers run as threads, and allows for code that is used by interrupt handlers, that would normally never schedule, to schedule.

### 2.2 Soft IRQs

Not only do hard interrupts run as threads, but all soft IRQs do as well. In the current mainline kernel,[7] soft IRQS are usually handled on exit of a hard interrupt. They can happen anytime interrupts and soft IRQs are enabled. Sometimes when a large number of soft IRQs need to be handled, they are pushed off to the ksoftirqd thread to complete them. But a soft IRQ handler can not assume that it will be running in a threaded context.

In the RT patch, the soft IRQs are only handled in a thread. Furthermore, they are split amongst several threads. Each soft IRQ has its own thread to handle them. This way, the system administrator can control the priority of individual soft IRQ threads.

Here's a snapshot of soft IRQ and hard IRQ threads, using ps -eo pid,pri,rtprio,cmd.

[6] see Section 4.

[7] 2.6.21 as the time of this writing.

| PID | PRI | RTPRIO | CMD |
|---|---|---|---|
| 4 | 90 | 50 | [softirq-high/0] |
| 5 | 90 | 50 | [softirq-timer/0] |
| 6 | 90 | 50 | [softirq-net-tx/] |
| 7 | 90 | 50 | [softirq-net-rx/] |
| 8 | 90 | 50 | [softirq-block/0] |
| 9 | 90 | 50 | [softirq-tasklet] |
| 10 | 90 | 50 | [softirq-sched/0] |
| 11 | 90 | 50 | [softirq-hrtimer] |
| 12 | 90 | 50 | [softirq-rcu/0] |
| 304 | 90 | 50 | [IRQ-8] |
| 347 | 90 | 50 | [IRQ-15] |
| 381 | 90 | 50 | [IRQ-12] |
| 382 | 90 | 50 | [IRQ-1] |
| 393 | 90 | 50 | [IRQ-4] |
| 400 | 90 | 50 | [IRQ-16] |
| 401 | 90 | 50 | [IRQ-18] |
| 402 | 90 | 50 | [IRQ-17] |
| 413 | 90 | 50 | [IRQ-19] |

## 3 Kernel Preemption

A critical section in the kernel is a series of operations that must be performed atomically. If a thread accesses a critical section while another thread is accessing it, data can be corrupted or the system may become unstable. Therefore, critical sections that can not be performed atomically by the hardware, must provide mutual exclusion to these areas. Mutual exclusion to a critical section may be implemented on a uniprocessor (UP) system by simply preventing the thread that accesses the section from being scheduled out (disable preemption). On a symmetric multiprocessor (SMP) system, disabling preemption is not enough. A thread on another CPU might access the critical section. On SMP systems, critical sections are also protected with locks.

An SMP system prevents concurrent access to a critical section by surrounding it with spin_locks. If one CPU has a thread accessing a critical section when another CPU's thread wants to access that same critical section, the second thread will perform a busy loop (spin) until the previous thread leaves that critical section.

A preemptive kernel must also protect those same critical sections from one thread accessing the section before another thread has left it. Preemption must be disabled while a thread is accessing a critical section, otherwise another thread may be scheduled and access that same critical section.

Linux, prior to the 2.5 kernel, was a non-preemptive kernel. That means that whenever a thread was running in kernel context (a user application making a system call) that thread would not be scheduled out unless it volunteered to schedule (calls the scheduler function). In the development of the 2.5 kernel, Robert Love introduced kernel preemption [2]. Robert Love realized that the critical sections that are protected by spin_locks for SMP systems, are also the same sections that must be protected from preemption. Love modified the kernel to allow preemption even when a thread is in kernel context. Love used the spin_locks to mark the critical sections that must disable preemption.[8]

The 2.6 Linux kernel has an option to enable kernel preemption. Kernel preemption has improved reaction time and lowered latencies. Although kernel preemption has brought Linux one step closer to an RTOS, Love's implementation contains a large bottleneck. A high priority process must still wait on a lower priority process while it is in a critical section, even if that same high priority process did not need to access that section.

## 4 Sleeping Spin Locks

Spin_locks are relatively fast. The idea behind a spin_lock is to protect critical sections that are very short. A spin_lock is considered fast compared to sleeping locks because it avoids the overhead of a schedule. If the time to run the code in a critical section is shorter than the time of a context switch, it is reasonable to use a spin_lock, and on contention, spin in a busy loop, while waiting for a thread on another CPU to release the spin_lock.

Since spin_locks may cause a thread on another CPU to enter a busy loop, extra care must be given with the use of spin_locks. A spin_lock that can be taken in interrupt context must always be taken with interrupts disabled. If an interrupt context handler that acquires a spin_lock is triggered while the current thread holds that same spin_lock, then the system will deadlock. The interrupt handler will spin on the spin_lock waiting for the lock to be released, but unfortunately, that same interrupt handler is preventing the thread that holds the spin_lock from releasing it.

A problem with the use of spin_locks in the Linux kernel is that they also protect large critical sections. With

[8] other areas must also be protected by preemption (e.g., interrupt context).

the use of nested spin_locks and large sections being protected by them, the latencies caused by spin_locks become a big problem for an RTOS. To address this, the RT patch converts most spin_locks into a mutex (sleeping lock).

By converting spin_locks into mutexes, the RT patch also enables preemption within these critical sections. If a thread tries to access a critical section while another thread is accessing it, it will now be scheduled out and sleep until the mutex protecting the critical section is released.

When the kernel is configured to have sleeping spin_locks, interrupt handlers must also be converted to threads since interrupt handlers also use spin_locks. Sleeping spin_locks are not compatible with not-threaded interrupts, since only a threaded interrupt may schedule. If a device interrupt handler uses a spin_lock and also sets the interrupt flag `IRQ_NODELAY`, the system will crash if the interrupt handler tries to acquire the spin_lock when it is already taken.

Some spin_locks in the RT kernel must remain a busy loop, and not be converted into a sleeping spin_lock. With the use of type definitions, the RT patch can magically convert nearly all spin_locks into sleeping mutexes, and leave other spin_locks alone.

## 5 The Spin Lock Maze

To avoid having to touch every spin_lock in the kernel, Ingo Molnar developed a way to use the latest gcc extensions to determine if a spin_lock should be used as a mutex, or stay as a busy loop.[9] There are places in the kernel that must still keep a true spin_lock, such as the scheduler and the implementation of mutexes themselves. When a spin_lock must remain a spin_lock the RT patch just needs to change the type of the spin_lock from `spinlock_t` to `raw_spinlock_t`. All the actual spin_lock function calls will determine at compile time which type of spin_lock should be used. If the spin_lock function's parameter is of the type `spinlock_t` it will become a mutex. If a spin_lock function's parameter is of the type `raw_spinlock_t` it will stay a busy loop (as well as disable preemption).

Looking into the header files of spinlock.h will drive a normal person mad. The macros defined in those headers are created to actually facilitate the code by not having to figure out whether a spin_lock function is for a mutex or a busy loop. The header files, unfortunately, are quite complex. To make it easier to understand, I will not show the actual macros that make up the spin_lock function, but, instead, I will show what it looks like evaluated slightly.

```
#define spin_lock(lock)
  if (TYPE_EQUAL((lock),
raw_spinlock_t))
     __spin_lock(lock);
  else if (TYPE_EQUAL((lock),
spinlock_t))
     _spin_lock(lock);
  else __bad_spinlock_type();
```

The `TYPE_EQUAL` is defined as `__builtin_types_compatible_p(typeof(lock), type *)` which is a gcc internal command that handles the condition at compile time. The `__bad_spinlock_type` function is not actually defined, if something other than a spinlock_t or raw_spinlock_t is passed to a spin_lock function the compiler will complain.

The `__spin_lock()`[10] acts like the original spin_lock function, and the `_spin_lock()`[11] evaluates to the mutex, `rt_spin_lock()`, defined in `kernel/rtmutex.c`.

## 6 Priority Inheritance

The most devastating latency that can occur in an RTOS is unbounded priority inversion. As mentioned earlier, priority inversion occurs when a high priority thread must wait on a lower priority thread before it can run. This usually occurs when a resource is shared between high and low priority threads, and the high priority thread needs to take the resource while the low priority thread holds it. Priority inversion is natural and can not be completely prevented. What we must prevent is unbounded priority inversion. That is when a high priority thread can wait an undetermined amount of time for the lower priority thread to release the resource.

The classic example of unbounded priority inversion takes place with three threads, each having a different priority. As shown in Figure 1, the CPU usage of three

[9] also called raw_spin_lock.

[10] prefixed with two underscores.

[11] prefixed with one underscore.

threads, A (highest priority) B (middle priority), and C (lowest priority). Thread C starts off and holds some lock, then thread A wakes up and preempts thread C. Thread A tries to take a resource that is held by thread C and is blocked. Thread C continues but is later preempted by thread B before thread C could release the resource that thread A is blocked on. Thread B is of higher priority than thread C but lower priority than thread A. By preempting thread C it is in essence preempting thread A. Since we have no idea how long thread B will run, thread A is now blocked for an undetermined amount of time. This is what is known as **unbounded priority inversion**.

There are different approaches to preventing unbounded priority inversion. One way is just simply by design. That is to carefully control what resources are shared as well as what threads can run at certain times. This is usually only feasible by small systems that can be completely audited for misbehaving threads. The Linux kernel is far too big and complex for this approach. Another approach is priority ceiling [3], where each resource (lock) knows the highest priority thread that will acquire it. When a thread takes a resource, it is temporarily boosted to the priority of that resource while it holds the resource. This prevents any other thread that might acquire that resource from preempting this thread. Since pretty much any resource or lock in the Linux kernel may be taken by any thread, you might as well just keep preemption off while a resource is held. This would include sleeping locks (mutexes) as well.

What the RT patch implements is **Priority Inheritance** (PI). This approach scales well with large projects, although it is usually criticized that the algorithms to implement PI are too complex and error prone. PI algorithms have matured and it is easier to audit the PI algorithm than the entire kernel. The basic idea of PI is that when a thread blocks on a resource that is owned by a lower priority thread, the lower priority thread inherits the priority of the blocked thread. This way the lower priority thread can not be preempted by threads that are of lower priority than the blocked thread. Figure 2 shows the same situation as Figure 1 but this time with PI implemented.

The priority inheritance algorithm used by the RT patch is indeed complex, but it has been vigorously tested and used in production environments. For a detailed explanation of the design of the PI algorithm used not only by the RT patch but also by the current mainline kernel PI futex, see the kernel source documentation [9].

## 7 What's Good for RT is Good for the Kernel

The problem with prior implementations of RT getting accepted into mainline Linux, was that too much was done independently from mainline development, or was focused strictly on the niche RT market. Large intrusive changes were made throughout the kernel in ways that were not acceptable by most of the kernel maintainers. Finally, one day Ingo Molnar noticed the benefits of RT and started developing a small project that would incorporate RT methods into the Linux kernel. Molnar, being a kernel maintainer, could look at RT from a more general point of view, and not just from that of a niche market. His approach was not to force the Linux kernel into the RT world, but rather to bring the beneficial parts of the RT world to Linux.

One of the largest problems back then (2004) was this nasty lock that was all over the Linux kernel. This lock is know as the Big Kernel Lock (BKL). The BKL was introduced into Linux as the first attempt to bring Linux to the multiprocessor environment. The BLK would protect concurrent accesses of critical sections from threads running on separate CPUs. The BKL was just one big lock to protect everything. Since then, spin_locks have been introduced to separate non related critical sections. But there are still large portions of Linux code that is still protected by the BKL.

The BKL was a spin_lock that was large and intrusive, and would cause large latencies. Not only was it a spinning lock, but it was also recursive.[12] It also had the non-intuitive characteristic that a process holding a BKL is allowed to voluntarily sleep. Spin locks could cause systems to lock up if a thread were to sleep while holding one, but the BKL was special, in that the scheduler magically released the BKL, and would reacquire the lock when the thread resumes.

Molnar developed a way to preempt this lock [6]. He changed the BKL from a spin lock into a mutex. To preserve the same semantics, the BKL would be released if the thread voluntarily scheduled, but not when the thread was preempted. Allowing threads to be preempted while holding the BKL greatly reduced the scheduling latencies of the kernel.

[12] Allowed the same owner to take it again while holding it, as long as it released the lock the same number of times.

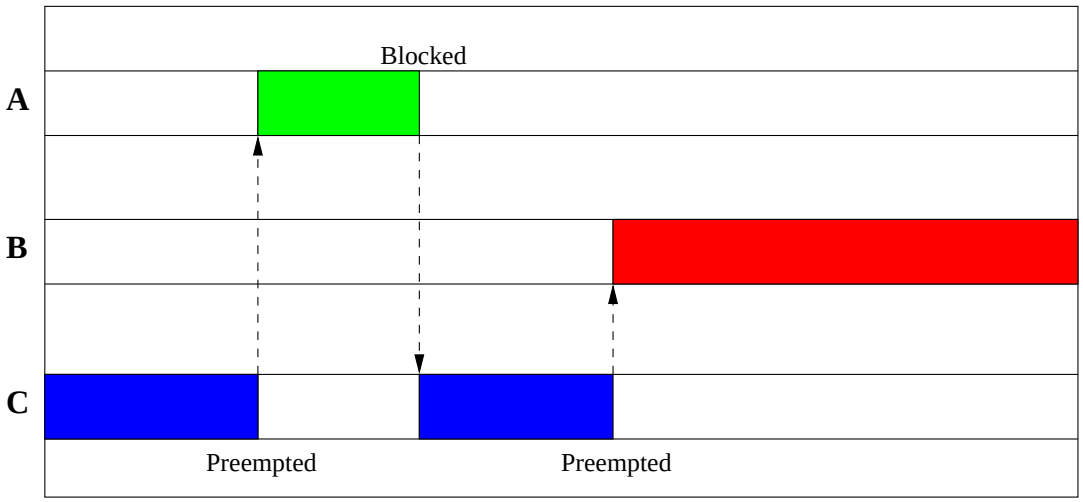


Figure 1: Priority Inversion

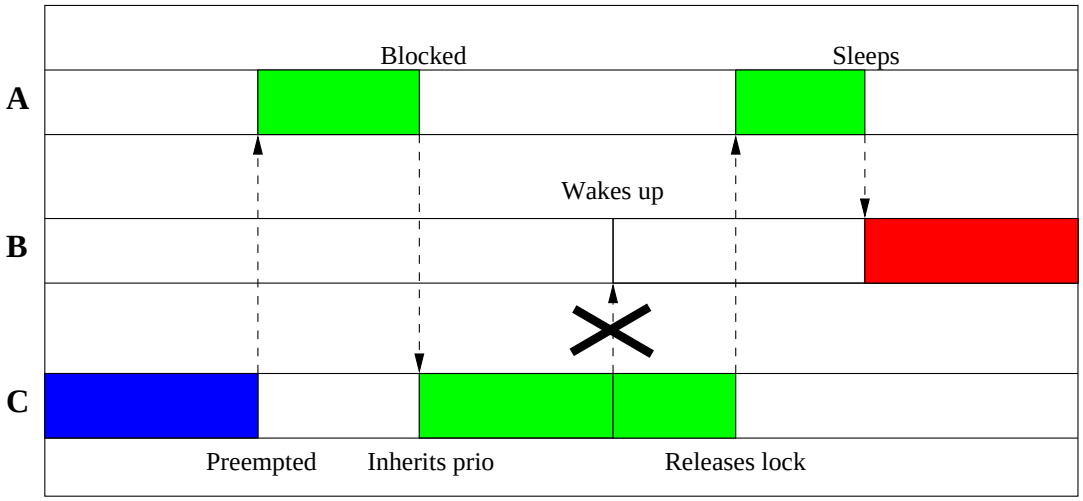


Figure 2: Priority Inheritance

### 7.1 Death of the Semaphore

Semaphores are powerful primitives that allow one or more threads access to a resource. The funny thing is, they were seldom used for multiple accesses. Most of the time they are used for maintaining mutual exclusion to a resource or to coordinate two or more threads. For coordination, one thread would down (lock) a semaphore and then wait on it.[13] Some other thread, after completing some task, would up (unlock) the semaphore to let the waiting threads know the task was completed.

[13] usually the semaphore was just created as locked.

The latter can be replaced by a `completion` [4]. A `completion` is a primitive that has the purpose of notifying one or more threads when another thread has completed an event. The use of semaphores for this purpose is now obsolete and should be replaced with `completions`.

There was nothing available in the mainline kernel to replace the semaphore for a single mutual exclusion lock. The ability of a semaphore to handle the case of multiple threads accessing a resource produces an overhead when only acting as a mutex. Most of the time this overhead is unnecessary. Molnar implemented a new primitive for the kernel called **mutex**. The simpler design of the mutex makes it much cleaner and slightly faster than a

semaphore. A mutex only allows a single thread access to a critical section, so it is simpler to implement and faster than a semaphore. So the addition of the mutex to the mainline kernel was a benefit for all. But Molnar had another motive for implementing the mutex (which most people could have guessed).

Semaphores, with the property of allowing more than one thread into a critical section, have no concept of an owner. Indeed, one thread may even release a semaphore while another thread acquires it.[14] A semaphore is never bound to a thread. A mutex, on the other hand, always has a one-to-one relationship with a thread when locked. The mutex may be owned by one, and only one, thread at a time. This is key to the RT kernel, as it is one of the requirements of the PI algorithm. That is, a lock may have one, and only one, owner. This ensures that a priority inheritance chain stays a single path, and does not branch with multiple lock owners needing to inherit a priority of a waiting thread.

### 7.2 The Path to RT

Every mainline release includes more code from the RT patch. Some of the changes are simply clean-ups and fixes for race conditions. An RTOS on a single CPU system exposes race conditions much easier than an 8 way SMP system. The race windows are larger due to the preemptive scheduling model. Although race conditions are easier to exposed on an RT kernel, those same race conditions still exist in the mainline kernel running on an SMP system. There have been arguments that some of the fixed race conditions have never been reported, so they most likely have never occurred. More likely, these race conditions have crashed some server somewhere, but since the race window is small, the crash is hard to reproduce. The crash would have been considered an anomaly and ignored. With the RT patch exposing rare bugs and the RT patch maintainers sending in fixes, the mainline kernel has become more stable with fewer of these seemingly unreproducable crashes.

In addition to clean-ups and bug fixes, several major features have been already incorporated into the mainline kernel.

- `gettimeofday (2.6.18)` – John Stultz's redesign of the time infrastructure.
- `Generic IRQS (2.6.18)` – Ingo Molnar's consolidation of the IRQ code to unify all the architectures.
- `High Resolution Timers Pt. 1 (2.6.16)` – Thomas Gleixner's separation of timers from timeouts.
- `High Resolution Timers Pt. 2 (2.6.21)` – Thomas Gleixner's change to the timer resolution. Now clock event resolution is no longer bound to jiffies, but to the underlining hardware itself.

One of the key features for incorporating the RT patch into mainline is PI. Linus Torvalds has previously stated that he would never allow PI to be incorporated into Linux. The PI algorithm used by the RT patch can also be used by user applications. Linux implements a user mutex that can create, acquire, and release the mutex lock completely in user space. This type of mutex is known as a **futex** (fast mutex) [1]. The futex only enters the kernel on contention. RT user applications require that the futex also implements PI and this is best done within the kernel. Torvalds allowed the PI algorithm to be incorporated into Linux (2.6.18), but only for the use with futexes.

Fortunately, the core PI algorithm that made it into the mainline Linux kernel is the same algorithm that is used by the RT patch itself. This is key to getting the rest of the RT patch upstream and also brings the RT patch closer to mainline, and facilitates the RT patch maintenance.

## 8 RT is the Gun to Shoot Yourself With

The RT patch is all about determinism and being able to have full control of the kernel. But, like being root, the more power you give the user the more likely they will destroy themselves with it. A common mistake for novice RT application developers is writing code like the following:

```
x = 0;
/* let another thread set x */
while (!x)
  sched_yield();
```

Running the above with the highest priority on an RTOS, and wondering why the system suddenly freezes.

[14] in the case of completions.

This paper is not about user applications and an RT kernel, but the focus is on developers working within the kernel and needing to understand the consequences of their code when someone configures in full RT.

### 8.1 yield() is Deadly

As with the above user land code, the kernel has a similar devil called `yield()`. Before using this, make sure you truly understand what it is that you are doing. There are only a few locations in the kernel that have legitimate uses of `yield()`. Remember in the RT kernel, even interrupts may be starved by some device driver thread looping on a `yield()`. `yield()` is usually related to something that can also be accomplished by implementing a `completion`.

Any kind of spinning loop is dangerous in an RTOS. Similar to using a busy loop spin_lock without disabling interrupts, and having that same lock used in an interrupt context handler, a spinning loop might starve the thread that will stop the loop. The following code that tries to prevent a reverse lock deadlock is no longer safe with the RT patch:

```
retry:
  spin_lock(A);
  if (!spin_trylock(B)) {
    spin_unlock(A);
    goto retry;
  }
```

Note: The code in `fs/jbd/commit.c` has such a situation.

### 8.2 rwlocks are Evil

Rwlocks are a favorite with many kernel developers. But there are consequences with using them. The rwlocks (for those that don't already know), allow multiple readers into a critical section and only one writer. A writer is only allowed in when no readers are accessing that area. Note, that rwlocks which are also implemented as busy loops on the current mainline kernel, are now sleeping mutexes in the RT kernel.[15]

Any type of read/write lock needs to be careful, since readers can starve out a writer, or writers can starve out the readers. But read/write locks are even more of a problem in the RT kernel. As explained in Section 6, PI is used to prevent unbounded priority inversion. Read/write locks do not have a concept of ownership. Multiple threads can read a critical section at the same time, and if a high priority writer were to need access, it would not be able to boost all the readers at that moment. The RT kernel is also about determinism, and known measurable latencies. The time a writer must wait, even if it were possible to boost all readers, would be the time the read lock is held multiplied by all the readers that currently have that lock.[16]

Presently, to solve this issue in the RT kernel, the rwlocks are not simply converted into a sleeping lock like spin_locks are. Read_locks are transformed into a recursive mutex so that two different threads can **not** enter a read section at the same time. But read_locks still remain recursive locks, meaning that the same thread can acquire the same read_lock multiple times as long as it releases the lock the same number of times it acquires it. So in the RT kernel, even read_locks are serialized among each other. RT is about predictable results, over performance. This is one of those cases that the overall performance of the system may suffer a little to keep guaranteed performance high.

### 8.3 Read Write seqlocks are Mischievous

As with rwlocks, read/write seqlocks can also cause a headache. These are not converted in the RT kernel. So it is even more important to understand the usage of these locks. Recently, the RT developers came across a regression with the introduction of some of the new time code that added more instances of the `xtime_lock`. This lock uses read/write seqlocks to protect it. The way the read/write seqlocks work, is that the read side enters a loop starting with `read_seqlock()` and ending with `read_sequnlock()`. If no writes occurred between the two, the `read_sequnlock()` returns zero, otherwise it returns non-zero. If something other than zero is returned by the `read_seqlock()`, the loop continues and the read is performed again.

The issue with the read/write seqlocks is that you can have multiple writes occur during the read seqlock. If the design of the seqlocks is not carefully thought out, you could starve the read lock. The situation with the

[15] mainline kernel also has sleeping rwlocks implemented with `up_read` and `down_read`.

[16] Some may currently be sleeping.

xtime_lock was even present in the 2.6.21-rc series. The xtime_lock should only be written to on one CPU, but a change that was made in the -rc series that allowed the xtime_lock to be written to on any CPU. Thus, one CPU could be reading the xtime_lock but all the other CPUs could be queueing up to write to it. Thus the latency of the read_seqlock is not only the time the read_seqlock is held, but also the sum of all the write_seqlocks that are run on each CPU. A poorly designed read/write seqlock implementation could even repeat the write_seqlocks for the CPUs. That is to say, while CPU 1 is doing the read_seqlock loop, CPU 2 does a write_seqlock, then CPU 3 does a write_seqlock, then CPU 4 does a write_seqlock, and by this time, CPU 2 is doing another write_seqlock. All along, leaving CPU 1 continually spinning on that read_seqlock.

### 8.4 Interrupt Handlers Are No Longer Supreme

Another gotcha for drivers that are running on the RT kernel is the assumption that the interrupt handler will occur when interrupts are enabled. As described in section threaded-interrupts, the interrupt service routines are now carried out with threads. This includes handlers that are run as soft IRQs (e.g., net-tx and net-rx). If a driver for some reason needs a service to go off periodically so that the device won't lockup, it can not rely on an interrupt or soft IRQ to go off at a reasonable time. There may be cases that the RT setup will have a thread at a higher priority than all the interrupt handlers. It is likely that this thread will run for a long period of time, and thus, starve out all interrupts.[17] If it is necessary for a device driver to periodically tickle the device then it must create its own kernel thread and put it up at the highest priority available.

## 9 Who Uses RT?

The RT patch has not only been around for development but there are also many users of it, and that number is constantly growing.

The audio folks found out that the RT patch has significantly helped them in their recordings (http://ccrma.stanford.edu/planetccrma/software).

IBM, Red Hat and Raytheon are bringing the RT patch to the Department of Defense (DoD). (http://www-03.ibm.com/press/us/en/pressrelease/21033.wss)

Financial institutions are expressing interest in using the RT kernel to ensure dependably consistent transaction times. This is increasingly important due to recently enacted trading regulations [8].

With the growing acceptance of the RT patch, it won't be long before the full patch is in the mainline kernel, and anyone can easily enjoy the enhancements that the RT patch brings to Linux.

## 10 RT Benchmarks

Darren V. Hart (dvhltc@us.ibm.com)

Determinism and latency are the key metrics used to discuss the suitability of a real-time operating system. IBM's Linux Technology Center has contributed several test cases and benchmarks which test these metrics in a number of ways. The results that follow are a small sampling that illustrates the features of the RT patch as well as the progress being made merging these features into the mainline Linux kernel. The tests were run on a 4 CPU Opteron system with a background load of make -j8 2.6.16 kernel build. Source for the tests used are linked to from the RT Community Wiki.[18] Full details of these results are available online [5].

### 10.1 gettimeofday() Latency

With their dependence on precise response times, real-time systems are prone to making numerous system calls to determine the current time. A deterministic implementation of gettimeofday() is critical. The gtod_latency test measures the difference between the time reported in pairs of consecutive gettimeofday() calls.

The scatter plots for 2.6.18 (Figure 3) and 2.6.18-rt7 (Figure 4) illustrate the reduced latency and improved determinism the RT patch brings to the gettimeofday() system call. The mainline kernel experiences a 208 us maximum latency, with a number of samples well above 20 us. Contrast that with the 17 us maximum latency of the 2.6.18-rt7 kernel (with the vast majority of samples under 10 us).

[17] besides the timer interrupt.

[18] http://rt.wiki.kernel.org/index.php/IBM_Test_Cases

### 10.2 Periodic Scheduling

Real-time systems often create high priority periodic threads. These threads perform a very small amount of work that must be performed at precise intervals. The results from the `sched_latency` measure the scheduling latency of a high priority periodic thread, with a 5 ms period.

Prior to the high resolution timer (hrtimers) work, timer latencies were only good to about three times the period of the periodic timer tick period (about 3ms with HZ=1000). This level of resolution makes accurate scheduling of periodic threads impractical since a task needing to be scheduled even a single microsecond after the timer tick would have to wait until the next tick, as illustrated in the latency histogram for 2.6.16 (Figure 5).[19] With the hrtimers patch included, the RT kernel demonstrates low microsecond accuracy (Figure 6), with a max scheduling latency of 25 us. The mainline 2.6.21 kernel has incorporated the hrtimers patch.

### 10.3 Asynchronous Event Handling

As discussed in Section 2, real-time systems depend on deterministic response times to asynchronous events. `async_handler` measures the latency of waking a thread waiting on an event. The events are generated using POSIX conditional variables.

Without the RT patch, the 2.6.20 kernel experiences a wide range of latencies while attempting to wake the event handler (Figure 7), with a standard deviation of 3.69 us. 2.6.20-rt8 improves on the mainline results, reducing the standard deviation to 1.16 us (Figure 8). While there is still some work to be done to reduce the maximum latency, the RT patch has greatly improved the deterministic behavior of asynchronous event handling.

## References

[1] Futex. `http://en.wikipedia.org/wiki/Futex`.

[2] Preemptible kernel patch makes it into linux kernel v2.5.4-pre6. `http://www.linuxdevices.com/news/NS3989618385.html`.

[3] Priority ceiling protocol. `http://en.wikipedia.org/wiki/Priority_ceiling_protocol`.

[4] Jonathan Corbet. Driver porting: completion events. `http://lwn.net/Articles/23993/`.

[5] Darren V. Hart. Ols 2007 - real-time linux latency comparisons. `http://www.kernel.org/pub/linux/kernel/people/dvhart/ols2007`.

[6] Ingo Molnar. `http://lwn.net/Articles/102216/`.

[7] Ingo Molnar. Rt patch. `http://people.redhat.com/mingo/realtime-preempt`.

[8] "The Trade News". Reg nms is driving broker-dealer investment in speed and storage technology. `http://www.thetradenews.com/regulation-compliance/compliance/671`.

[9] Steven Rostedt. Rt mutex design. `Documentation/rt-mutex-design.txt`.

[19] 2.6.16 was used as later mainline kernels were unable to complete the test.

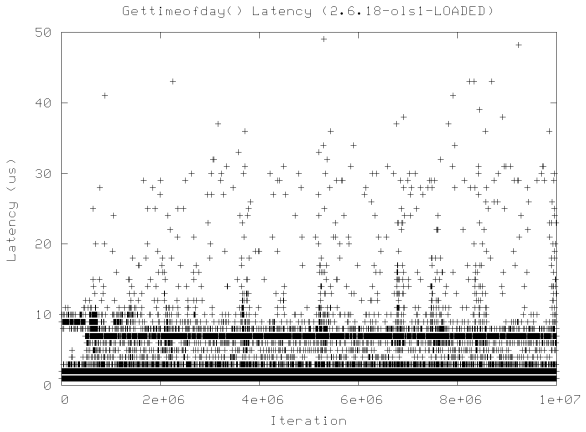


Figure 3: 2.6.18 gtod_latency scatter plot

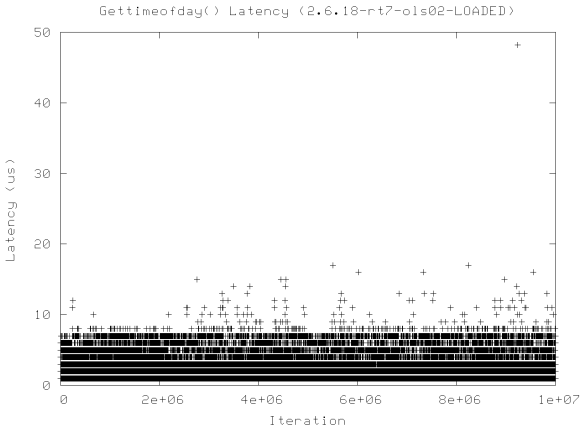


Figure 4: 2.6.18-rt7 gtod_latency scatter plot

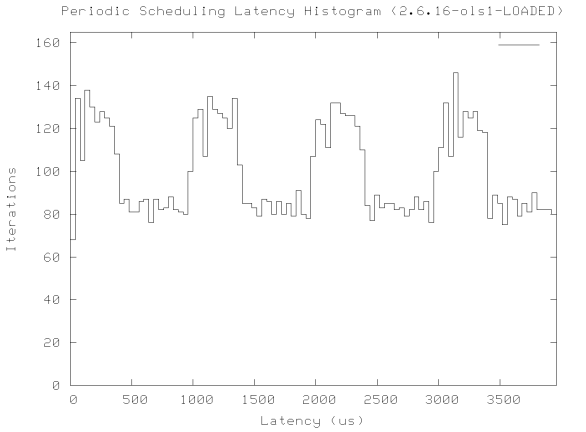


Figure 5: 2.6.16 sched_latency histogram

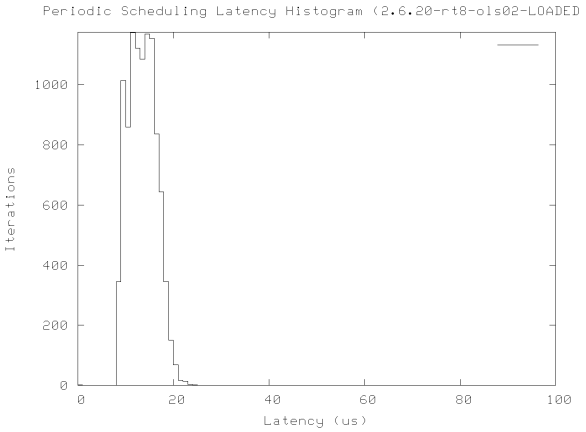


Figure 6: 2.6.20-rt8 sched_latency histogram

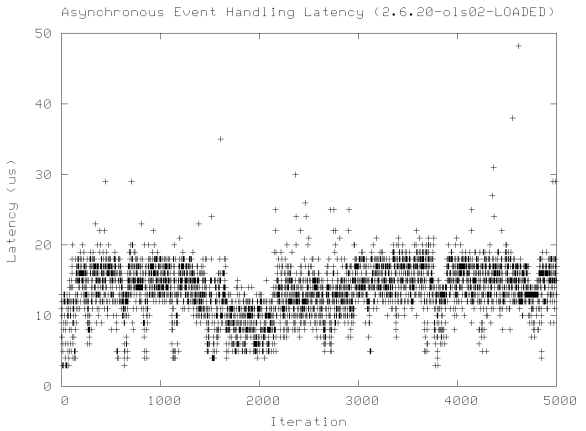


Figure 7: 2.6.20 async_handler scatter plot

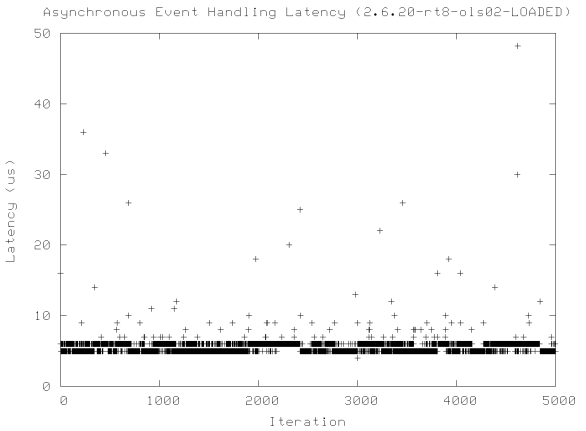


Figure 8: 2.6.20-rt8 async_handler scatter plot

# Proceedings of the Linux Symposium

Volume Two

June 27th–30th, 2007
Ottawa, Ontario
Canada

## Conference Organizers

Andrew J. Hutton, *Steamballoon, Inc., Linux Symposium, Thin Lines Mountaineering*
C. Craig Ross, *Linux Symposium*

## Review Committee

Andrew J. Hutton, *Steamballoon, Inc., Linux Symposium, Thin Lines Mountaineering*
Dirk Hohndel, *Intel*
Martin Bligh, *Google*
Gerrit Huizenga, *IBM*
Dave Jones, *Red Hat, Inc.*
C. Craig Ross, *Linux Symposium*

## Proceedings Formatting Team

John W. Lockhart, *Red Hat, Inc.*
Gurhan Ozen, *Red Hat, Inc.*
John Feeney, *Red Hat, Inc.*
Len DiMaggio, *Red Hat, Inc.*
John Poelstra, *Red Hat, Inc.*