The Evolution of Real-Time Linux

Sven-Thorsten Dietrich
MontaVista Software, Inc
1237 East Arques Avenue, Sunnyvale, CA, USA
sven@mvista.com

Daniel Walker
MontaVista Software, Inc
1237 East Arques Avenue, Sunnyvale, CA, USA
dwalker@mvista.com

Abstract

In October, 2004 the authors of this paper announced a Real-Time Linux Kernel prototype on the Linux Kernel mailing list. The Real-Time Linux prototype introduced preemptible locking into the Linux kernel, and allowed task preemption to occur while tasks were executing within critical sections, resulting in a dramatic improvement in the Real-Time response of the Linux kernel.

Ingo Molnar reviewed the preemptible locking prototype and quickly implemented a preemptible kernel on top of his existing Voluntary Preemption project. This effort gave rise to the Real-Time preemption patch, which is currently available for download:

http://people.redhat.com/~mingo/realtime-preempt/

This paper examines the foundations of the Real-Time Linux kernel, focusing in detail on [1] thread-context interrupt handling, [2] replacement of non-preemptible locking with preemptible mutex-based locks, [3] priority-inheritance, [4] virtualization of interrupt-disable, [5] integration of high-resolution timers, as well as current development on [6] user-space extensions of the kernel real-time mutex.

1 Introduction: Kernel Real-Time Support

The processing of time-critical tasks, depends upon the capacity of the system to respond to an event, within a known and bounded interval of time.

Our objective is to enable the Linux 2.6 kernel to be usable for high-performance multi-media applications and for applications requiring very fast, reliable task-level control functions.

The AV industry is building HDTV related technology on Linux, and high-end desktop systems are used by audio enthusiasts for real-time effects.

Cell phones, PDAs and MP3 players are converging into highly integrated devices requiring a large number of threads. These threads support a vast array of communications protocols (IP, Bluetooth, 802.11, GSM, CDMA, etc.). Especially the cellular-based protocols require highly deadline-sensitive operations to work reliably.

Linux is being increasingly utilized in traditional real-time control environments including radar processing, factory automation systems, "in the loop" process control systems, medical and instrumentation systems, and automotive control systems. Many times these systems have task level response requirements in the 10's to hundreds of microsecond ranges, which is a level of guaranteed task response not achievable with current 2.6 Linux technology.

1.1 Precedent Work

There are several micro-kernel solutions available, which achieve the required performance. The use of these systems invariably requires a kernel patch, but there are two separate kernel environments, creating more overall system, and application-design complexity.

Significant precedent work has also been performed on the 2.4 Linux kernel under the auspices of the Kansas University RealTime (KURT) Linux Project, directed by Prof. Douglas Niehaus."

http://www.cs.wustl.edu/~niehaus/kurt.html

This project has recently been renamed KUSP (Kansas University System Programming), and is currently focusing on research pertaining to:

- modular hierarchical scheduling
- high accurate time synchronization for distributed systems
- modular and configurable instrumentation

http://kusp.ittc.ku.edu

2 Kernel Preemption Background

Task preemption has been available in all versions of Linux. Linux kernels have continuously become more sophisticated in their flexibility to preempt tasks executing within the kernel.

The response time of the Linux kernel has generally improved over time, while continuously becoming more versatile, and finding a very broad range of application hosts.

With the 2.4 Kernel Series, Linux became fully capable of meeting high-performance, multi-media streaming requirements at both the UI level, as well as on the infrastructure, or network carrier level.

See Table 1 for a summary of the general preemption capabilities of the Linux Kernel over time.

2.1 Early Linux Kernels (Linux 1.x and Linux 2.0.x / 2.2.x)

In early kernels, tasks could not be preempted once they started executing code in the kernel or in a driver. Scheduling could take place after the task voluntarily suspended itself in the kernel, or preemption could occur when the task exited the Kernel.

2.2 SMP Kernels (Linux 2.x)

The Linux 2.x kernel series introduced SMP support.

The earliest SMP Linux design was problematic, because CPUs could sit idle for long periods of time, while task on another CPU executed in the kernel.

Gradual improvements in locking technology, and especially decoupling of locks led towards continuous improvement, moving away from a single locking bottleneck (the Big-Kernel-Lock), towards data-specific, distributed locking of distinct critical sections.

Multiple threads could execute in the kernel as long as they did not attempt to access the same shared data at the same time. If multiple tasks contended for the same critical section, the task arriving later would have to wait.

The final SMP design in Late 2.4 Kernels used decoupled critical sections throughout the kernel, and dramatically improved the throughput of SMP systems.

The global Big-Kernel lock, still remained as a non-preemptible monolith, and continues to be associated with significant sources of preemption latency.

2.3 Preemptible Kernel: (Linux 2.4 + Preemption Patch)

In 2001, MontaVista began to enhance the evolving SMP locking paradigm to allow in-Kernel-preemption on uniprocessor systems.

The innovated preemption concept recognized that tasks could be preempted anywhere in the Kernel, as long as they were not executing in a critical section bounded by SMP locking.

The conceptual logic assumes, that multiple concurrent tasks could already execute in the kernel on SMP systems, and therefore, multiple tasks could also execute in the Kernel at the same time, on a uniprocessor system.

The preemption enhancements significantly increased the preemptible code surface within the Kernel, and improved the response time dramatically.

Massive worst case latency scenarios had been eliminated.

During its prime, the late Linux 2.4 Kernel outperformed any other major general purpose operating system in multimedia application reliability.

2.4 Preemptible Linux 2.6 Kernel

The advent of the 2.6 kernel series brought to mainstream many improvements and features that evolved from the requirements of embedded systems, including a new scheduler and in-kernel preemption.

Expectations on the real-time performance of the Linux 2.6 kernel were running high, based on the advancements in scheduler technology, as well as preemption enhancements.

The new scheduler performed nicely, but response time in the early 2.6 kernel continued to be impacted by preemption delays encountered when running tasks accessed shared system data inside a critical section.

Efforts have been underway for years, to reduce the number and duration of critical sections in the Linux kernel, as well as to reduce the nesting and locking durations of individual critical sections.

Kernel Version	Preemption Capability
Early Linux Kernel 1.x	No In-Kernel preemption
SMP Linux Kernel 2.x	No In-Kernel preemption
	BKL SMP Lock
SMP Linux Kernel 2.2 2.4	No In-Kernel preemption
	Spin-locked Critical Sections
Preemptible Linux Kernel 2.4	Preemptible
	Non-Preemptible Spinlock Sections
Current Linux Kernel 2.6.x	Preemptible
	Non-Preemptible Spin-lock Sections
	Preemptible BKL (since 2.6.11)
RT-Preempt Linux Kernel 2.6.x	Preemptible Kernel Critical Sections
	Preemptible IRQ Subsystem
	Mutex Locks with Priority Inheritance

Table 1: Preemption Capabilities by Kernel Version

Various forms of Instrumentation have been developed to identify the longest critical sections. That instrumentation generally applied to the Linux 2.6 kernel, and recent additional work was performed by Ingo Molnar, to develop more sophisticated latency-analysis instrumentation (also known as Latency tracing).

Initial measurements indicated that the preemption performance of the 2.6 kernel lagged the performance of the 2.4 kernel.

For audio processing, the 2.4 kernel series also outperformed some revisions of the early 2.6 Linux kernel series.

2.5 The Audio Community and Preemption Requirements

The Linux Audio community has always had stringent requirements for Linux task response time, due to the ever-advancing performance of audio applications, which are operating on streaming digital data, in real-time.

One of the most significant efforts by the audio community, to communicate the shortcomings of the Linux kernel to Linus, and the kernel developer community, was authored by Paul Barton-Davis and Benno Senoner.

This communication was titled "a joint letter on low latency and Linux" [2]

The document was signed by a large number of audio users and developers, who were attempting to raised the issue of Linux Kernel preemption latency to a new level.

"Their request was based on their desire to have Linux function well with audio, music and MIDI. Senoner produced some benchmarking software that demonstrated that the 2.2 kernel (and later the 2.4 kernel) had worst-case preemption latencies on the order of 100ms (http://www.gardena.net/benno/linux/audio/). Latencies of this magnitude are unacceptable for audio applications. Conventional wisdom seems to say that latencies on the order of no more than a few milliseconds are required." [3]

"Two efforts emerged that produced patched kernels that provided quite reasonable preemption latencies. Ingo Molnar (of Red Hat) and Andrew Morton (then of The University of Wollongong) both produced patch sets that provided preemption within particularly long sections in the kernel. You can find Ingo Molnar's patches at http://people.redhat.com/mingo/lowlatency-patches/, and you can find Andrew Morton's work at http://www.zipworld.com.au/~akpm/linux/schedlat.html" [3]

The audio community continues to be active in its interactions with the Kernel developer community, and has been instrumental in supporting the Real-Time preemption technology discussed in this paper.

The following section examines the Kernel constructs contributing to preemption latency, in order to establish a foundation for discussion of the Real-Time preemption enhancements.

3 Concurrency in the Uniprocessor Kernel

Uniproecssor(UP)：单处理器

This section examines critical section management and task preemption control in the Linux 2.6 kernel.

3.1 Critical Sections

During the Kernel's operation, shared Kernel data objects, as well as buses and other hardware reg-

isters, are subject to concurrent read-modify-write access by multiple threads and interrupt handlers.

Shared data that is operated upon by multiple independent tasks, must be protected from context switches such that read-modify-update operations are not overlapped

In order to guarantee data integrity, transactionalized accesses are imposed on all code operating on Kernel data objects, by means of critical sections.

A critical section estabishes a section of protected code, wherein only one task is allowed to operate.

Any process executing within the boundaries of a critical section, must exit the critical section before another process is allowed to enter.

The critical sections boundaries serialize concurrent accesses to shared objects and therefore, guarantee that shared data and hardware register operations remain in a stable state at all times.

There are two basic classes of critical sections in the kernel:

1. Critical Sections protecting threads from concurrent access by other threads
2. Critical Sections protecting threads from concurrent access by interrupts

3.2 Critical Section Management

Critical section management in a Linux uniprocessor environment is the most basic and serves as the entry point into the discussion. In the UP environment, only a single thread of execution exists on the CPU. The running (current) thread executes until it relinquises the CPU voluntarily, or is interrupted by a processor exception, such as an interrupt.

3.3 Protection from other Threads

The simplest form of critical section available in the Linux kernel is known as a preemption-disabled section of code. Concurrency protection from other threads (but not interrupts) is achieved by suppressing task switches while a thread is executing inside a critical section. Task switches are suppressed by the preempt_disable function. This Kernel design-feature guarantees, that a thread can transition the critical section before another thread is allowed to execute. The requirements of the critical section are fulfilled, since entry into a simple critical section using the preempt_disable() primitive, establishes a non-preemptible thread execution state.

```
#define preempt_disable() \
do { \
        inc_preempt_count(); \
        barrier(); \
} while (0)
```

Each Linux task has a preemption counter, preempt_count, in its task control structure. The preempt_disable() function increments the threads preempt_count variable. The preempt_count variable is allocated on a per-thread basis in the asm-<arch>/thread_info.h header file, but the value of preempt_count exerts a global effect upon the system:

If preempt_count is non-zero, task switches are prohibited on the CPU. preempt_enable() decrements the preempt_count variable, via the preempt_enable_no_resched() call.

```
#define preempt_enable() \
do { \
        preempt_enable_no_resched(); \
        preempt_check_resched(); \
} while (0)
```

Preemption may only tke place only if preempt_count is equal to zero.

3.3.1 Nesting of Critical Sections

It is possible for critical sections to be nested inside the kernel. In this case, each successive critical section entered by the executing thread, is associated with another call to preempt_disable(). The preempt_count variable tracks the nesting of critical sections, and must be decremented by a call to preempt_enable(), when a critical section is unlocked. Only after the last critical section has been unlocked, does the value of preempt_count reach 0, indicating that the task may be preempted. In order to achieve proper accounting of critical section nesting, each call to preempt_enable(), must thus be balanced by a subsequent call to preempt_disable(). Rid can check it.

3.4 Protection from Interrupt Handlers

Critical sections demarcated by preempt_disable() and preempt_enable() pairs run with interrupts enabled. If an interrupt occurs while a thread is executing in a critical section, exception processing proceeds immediately, and the thread of execution is always returned to the running thread. The current threads preempt_count is sampled upon return from the exception, and that state determines, whether the return from exception passes through the scheduler, or continues with the interrupted thread. The code for handling interrupts while preemption is disabled, is typically found in assembly code in the interrupt dispatch routines located in the architecture-specific entry.S file.

这个section大意是说preempt_disable()只保证临界区代码不会被其它线程干扰，但是不保证不会被中断干扰，想避免中断抢占临界区，需要用local_irq_disable()。

If a data object is shared with an interrupt handler, it thus does not suffice to disable preemption, because the processors hardware exception mechanism does not understand the meaning of the thread's preempt_count variable.

Concurrency protection for data shared with interrupt handlers, is managed by disabling interrupts using the Kernels local_irq_disable() function.

Before a thread accesses data shared with an interrupt handler, interrupts are disabled, and they are not re-enabled until the data update has been completed. The interrupt-disable protects data from concurrent access by the interrupt handler, after the executing thread has entered the critical section.

3.5 Protection from Threads and Interrupt Handlers

If the shared data object is accessed by multiple threads as well as interrupt handler(s), it is necessary to disable preemption, as well as interrupts, to satisfy the protection requirements of the data object within the Kernel.

3.6 Critical Section Independence

The preempt-disable and interrupt-disable mechanisms are sufficient for basic critical section management in a UP environment, where only a single, active thread of execution exists.

It is important to observe, that the uniprocessor Linux kernel, manages critical sections, by categorically disabling task switches throughout the system.

When a task locks a critical section, preemption is disabled, and no other task can run. Preventing other tasks from running also prevents other tasks from accessing critical sections.

When a locked critical section is unlocked, the thread previously executing in the critical section may be preempted, if its preempt_count is equal to 0.

The Linux Kernel is designed around the assumption that the scheduler will not be called to reschedule a thread, while the threads preempt_count is non-zero.

Consequently, all suspended (preempted) threads must have a preempt_count of 0 stored. The per-thread allocation of the preempt_count variable, is therefore not necessary on the uniprocessor.

The per-thread preempt_count exists to provide a consistent implementation environment between uniprocessor and SMP systems.

4 Concurrency in the SMP Kernel

Concurrency management in the SMP Linux Kernel is significantly more complex, than in the uniprocessor Kernel.

4.1 Critical Sections

In an SMP environment, is not sufficient to disable preemption and interrupts, to protect data shared between CPUs.

Two, or more independent tasks, executing on different CPUs, could enter the same critical section at the same time, having no knowledge, whether preemption and interrupts are disabled on another CPU.

To protect shared data in the Kernel from concurrent access by multiple CPUs, it is therefore necessary, to physically allocate shared locks that control access to critical sections.

4.2 Critical Section Management

In the SMP Kernel, the data structures which protect access to critical sections from concurrent tasks on multiple CPUs, are known as **spinlocks**.

Unique spinlocks are allocated in memory, and identify discrete critical sections associated with specific shared data.

The SMP spinlock architecture decouples locking for independent data. Independent data objects can be locked independently, and allow multiple threads to execute in multiple critical sections of the kernel at the same time.

4.2.1 Uniprocessor Spin Lock Definition

The code between SMP and UP Kernels is common, and for this reason, the spinlock operations are found throughout the Kernel.

When CONFIG_SMP is not defined at build time, the implementation uses the uniprocessor definition of spinlocks

```
typedef struct { } spinlock_t;
```

The uniprocessor implementation of spinlock does not allocate any physical memory.

A side-effect of this optimization is, that the identity of the locked critical section is sacrificed by the same UP optimization. The UP- optimized spinlock implementation does not provide any means to identify, whether a given critical section is locked or unlocked, without examining the running task's program counter, or compiling in additional information via the DEBUG_SPINLOCK config option.

4.2.2 SMP Spin Lock Definition

Critical sections in the SMP Kernel are demarcated by physical spinlocks:

```
typedef struct {
        volatile unsigned int lock;
} spinlock_t;
```

4.3 Spin Lock Operation

When a task acquires a spinlock:

- preemption is disabled
- the acquiring task is allowed into the critical section
- No task switches can occur on the processor, until a spin_unlock operation takes place
- Interrupts may be disabled, if a function from spin_lock_irq family has been called

A spinlock is acquired at the beginning of a critical section, via the spin_lock operation.

```
#define spin_lock(lock) _spin_lock(lock)

#define _spin_lock(lock)        \
do { \
     preempt_disable(); \
     _raw_spin_lock(lock); \
} while(0)

# define _raw_spin_lock(lock)
     __raw_spin_lock(&(lock)->raw_lock)
```

Rescheduling can take place at the end of a critical section, when the spinlock is unlocked by the preempt_enable() operation, which as called as part of the spin_unlock operation:

```
#define spin_unlock(lock) _spin_unlock(lock)

#define _spin_unlock(lock) \
do { \
     _raw_spin_unlock(lock); \
     preempt_enable(); \
} while (0)

#define _raw_spin_unlock(lock)
     __raw_spin_unlock(&(lock)->raw_lock)
```

The _raw_spin_lock and _raw_spin_unlock functions establish the abstraction layer accommodating the necessary, architecture-specific, physical instruction implementation.

In addition, the _raw_spin_lock implementation is also different for SMP and UP configurations on each architecture.

4.3.1 Uniprocressor Spin Lock Operation

On uniprocessor systems, the implementation of _raw_spin_lock and _raw_spin_unlock, are defined as a no-op:

```
#define __raw_spin_lock(lock) \
               do { (void)(lock); } while(0)

#define __raw_spin_unlock(lock) \
               do { (void)(lock); } while(0)
```

In the uniprocessor kernel spinlocks thus degenerate to simple preempt_disable / preempt_enable operations.

The preempt_count variable is essential to correctly track the preemption state across nested critical sections in a UP system.

4.3.2 SMP Spin Lock Operation

The locking operation is significantly more complex on SMP, since the lock must be physically manipulated in a specific memory transaction required to obtain the lock.

The lock operation is typically performed by special processor instructions, known as test-and-set, or compare-and-exchange operations. The special atomic instructions guarantee completion of a memory operation in a single update cycle, such that no hardware-level races can ensue.

The architecture-specific implementation of the atomic __raw_spin_lock operation, guarantees that no other CPU can modify the same memory at the same time and obtain the same result.

Shown is the i386 implementation of the __raw operations:

```
static inline void
     __raw_spin_lock(raw_spinlock_t *lock)
{
     __asm__ __volatile__(
             __raw_spin_lock_string
             :"=m" (lock->slock) : : "memory");
}

#define __raw_spin_lock_string \
        "\n1:\t" \
        "lock ; decb %0\n\t" \
        "jns 3f\n" \
        "2:\t" \
        "rep;nop\n\t" \
        "cmpb $0,%0\n\t" \
        "jle 2b\n\t" \
        "jmp 1b\n" \
        "3:\n\t"
```

In the SMP kernel implementation of spin_unlock, an architecture-specific _raw_spin_unlock operation, similar to the _raw_spin_lock operation, performs the architecture-specific atomic memory operation.

```
static inline void
     __raw_spin_unlock(raw_spinlock_t *lock)
{
     __asm__ __volatile__(
             __raw_spin_unlock_string
     );
}

#define __raw_spin_unlock_string \
     "movb $1,%0" \
          :"=m" (lock->slock) : : "memory"
```

4.4 Spinlock Contention

Critical sections protected by spinlocks are non-preemptible.

4.4.1 Protection from other Threads

If a thread encounters a spinlock in the SMP kernel on an SMP system, then a lock contention situation arises, where a task on another CPU is holding the lock. The thread contending for the locked spinlock, will busy-wait, or "spin", while waiting for the lock to be released. This behavior gives rise to the name "spin_lock".

While a thread is "spinning" on a CPU, waiting for a lock, no useful work is being performed on that CPU.

4.4.2 Protection from Interrupts and Threads

Protection of shared data from threads and interrupts is conceptually similar to the uniprocessor implementation: In addition to disabling preemption on the local CPU, and acquiring a spinlock, interrupts must also be disabled.

A special set of variants of the spin_lock family of functions addresses this scenario:

The spin_lock_irq functions allow spinlocks to protect code shared with exception handlers.

As discussed previously, data shared with interrupt handlers must be protected by disabling interrupts, to avoid corruption by an interrupt. In addition, data protected by a critical section, can cause a dead lock if the exception occurs in a critical section and contends for a lock that is already held by the task on that CPU. In this event, the interrupt handler would spin, waiting for the lock to be released, while blocking the task holding the lock from running.

The spin_lock functions which also disable interrupts are:

- spin_lock_irq
- spin_lock_irqsave

4.5 Summary

The implementation of spinlocks extends the functionality of preempt_disable/preempt_enable() to the SMP Kernel.

In the UP kernel, the identity of the spinlock protected critical section is lost at compile time. It is impossible to identify which critical section is locked in a uniprocessor kernel, without knowledge of the program counter.

The locked critical section is readily identifiable (in SMP configuration), since the spinlock associated with that section will be locked.

The SMP implementation is more relevant to the fully-preemptible Real-Time Kernel, since any SMP-safe code in the kernel is also, by definition, fully preemptible.

5 Kernel Preemption Latency Analysis

The preemption latency of the Linux 2.6 Kernel is composed of two general classes:

1. Latencies caused by interrupt-domain processing
2. Latencies caused by critical-section processing

5.1 Interrupt-Domain Latencies

In the Linux kernel, it is possible for a thread to be suspended while executing, for the duration of interrupt exception processing. No task switches can occur until interrupt processing has completed.

Preemption latency associated with interrupt-domain processing has 3 sub-components:

1. Interrupt-disabled code sections. The IRQ-disable sections in the code have a minor impact on total task preemption latency.
2. Interrupt-context processing of system and driver interrupts is a major contributor to task preemption latency. In Linux, interrupt-context processing takes place at priorities

above task priority. Preemption latency is especially affected when heavy I/O transients are encountered.

3. Interrupt-context softIRQ processing is a major contributor to task preemption latency. Network-related softIRQ processing has been identified as a specific source of significant preemption latency.

It is possible to move 100% of softIRQ processing activity into softirqd, reducing the impact of softIRQ processing on preemption latency. No other, significant opportunies for improvement of preemption latency response time exists without extending the current design of the interrupt-handler domain.

5.2 Task-Domain Latencies

The Linux kernel disables preemption while a critical-section operation is underway. Consequently, a high priority task can be blocked from preempting any lower priority task in a critical section, until the low priority task exits the critical section and re-enables preemption.

Preemption latency associated with critical section processing is governed by the following Linux Kernel design features:

1. Under the non-preemptible locking design of the Linux 2.6 Kernel, any critical section can block preemption, and therefore suppress scheduling and execution of a high priority task. If a thread is holding a spinlock, any other task attempting to access the lock must wait for the lock to be unlocked. A high-priority process will therefore experience variable, and potentially extended delays, resulting from unpredictable preemption latencies associated with locked critical sections.

2. In the SMP Kernel, interrupt activity can extend the locking time of non-preemptible spinlocks. If interrupts are enabled on the CPU, where a thread is executing in a critical section, and another thread on another CPU contends for the critical section, then that thread will incur additional latency in obtaining the spinlock. If interrupt activity is taking place on the CPU owning the spinlock, then the interrupts delay the thread owning the lock, and therefore also delay the thread waiting for the lock.

3. Swapping preempts all running processes, while paging virtual memory into RAM. The kernel swap daemon, kswapd, runs at "uncontrollable" priority levels as a result of page-fault exceptions. Swapping can cause priority inversion, since high-priority processes can be blocked by swapping activity on behalf of a low priority process.

4. In the UP kernel, locking any critical section is equivalent to locking all critical sections, regardless of whether they are accessed or not. For example, even if it is known, that a waiting high-priority task does not access any critical sections, and thus poses no risk of corrupting shared data, the current spinlock implementation blocks execution of that task.

5. The BKL (Big Kernel Lock) is a relic from the early 2.x kernels series. When a task locks the BKL, preemption is disabled. The BKL is associated with locking during module loading and unloading, and use of the BKL has not been completely eliminated elsewhere in the Kernel.

   The use of the Big Kernel Lock is unique in the Kernel, since this lock can be relinquished by a task suspending itself, and is subsequently re-acquired by the same task, before proceding. The design of the BKL creates an SMP bottleneck, and therefore its use is deprecated.

   In the 2.6.11 Kernel, the BKL has been converted to a system semaphore, allowing SMP tasks to block, rather than to spin on the CPU, waiting for another CPU's task to release the BKL. This transition reducing the BKL's impact on SMP throughput, and improves preemption latency in the Kernel.

5.3 Aggregate Preemption Latency in Linux

The worst-case preemption latency is defined as the aggregate of worst-case interrupt-domain processing latency in the kernel, combined with the worst-case path through any nesting of critical sections.

The connection between interrupt-domain latencies and task-level preemption latencies requires that improvements be made across all Kernel subsystems affecting preemption latency, in order to achieve a consistent performance improvement.

The locking architecture of the Linux Kernel offers anly very limited, but labor-intensive potential for improvement of worst-case preemption latencies.

Eventually, there would be a plateau of relatively constant preemption latency, but that level would still be too high for a large class of real-time applications.

Nevertheless, due to the impact of priority inversion, it would still be possible for general high-priority system applications, to incur excessive latencies that could maliciously disrupt the application's time-critical operation.

5.4 Summary

- There are over 10,000 individual critical sections in a typical Linux 2.6 Kernel. Identifying the longest critical sections is a time-consuming process based on extensive testing and analysis.
- Modifying the longest critical sections to conform to a maximum is tedious and error-prone. Optimization of the performance of the Kernel's non-preemptible regions, has been compared to leveling a mountain range by scraping away from the tops
- Once a performance improvement is achieved, maintaining Kernel critical sections to a maximum execution time standard required continuous regression-tracking effort
- The existing locking-architecture constrains SMP scalability as well as preemption performance. This is especially relevant in near-future n-way systems based on dual, quad, and 8-way cores planned by semiconductor manufacturers.

5.5 Conclusion

1. Significant interrupt-domain enhancements pertaining to preemption latencies are are not possible in the Community Kernel's existing interrupt subsystem.
2. It is prohibitive to optimize non-preemptible critical sections on a case-by case basis to improve task-domain preemption.

6 The Fully-Preemptible Real-Time Kernel

Realization of significant preemption performance improvements in the Linux Kernel mandates re-examination of the original Linux design principles, as they pertain to interrupt processing and critical section management.

The Real-Time enhancements maintained in Ingo Molnar's patch apply to the current Kernel shortcomings in regards to aggregate preemption latency, without compromising the locking principles established in the Community kernel.

6.1 Requirements

One of the primary principles of Linux kernel development, is to produce generic kernel code, that can easily be ported to architectures, and functions on the widest-possible range of installed systems.

The general objectives for the real-time preemption implementation can be outlined as follows:

- Minimize the reliance upon architecture-specific hardware features
- Readily portable to new architectures
- Binary-compatibility with Community Linux kernels
- Unmodified, but SMP-correct, and up-to-date, 3rd-party drivers must compile and work in the RT kernel

The real-time preemption enhancements achieve all of the criteria, offering a very portable, yet aggressive solution to the Kernel′s preemption deficiencies.

The enhancements to the Community Linux kernel, necessary to achieve real-time performance are structured to address the specific classes of shortcomings in the Community Kernel:

1. Interrupt-domain enhancements to reduce latencies caused by Interrupt processing.
2. Critical section locking enhancements to reduce task level latencies caused by locking of critical sections within the Kernel to protect shared data.
3. Further decoupling of shared data into per-cpu variable and independent critical sections.

6.2 Thread-Context Interrupt Handling

The real-time Kernel demotes all interrupt activity from interrupt-context, and create tasks to handle interrupts in task context.

This approach allows a generic solution to be provided, that is portable, and independent of architecture-specific enhanced capabilities.

6.2.1 Interrupt Priorities

The following functional requirements are imposed on the real-time interrupt subsystem:

1. Interrupt execution must be preemptible
2. Interrupt execution must adhere priorities
3. Interrupt priories must be allocatable in task-level priority space, and it must be possible to elevate a real-time task′s priority above interrupt handling
4. It should be possible to reproduce hardware-level interrupt priority assignments with task-level interrupt priority assignments

Hardware priority support is not available on all Linux architectures, prohibiting generic solutions that consistently leverage hardware-interrupt priority.

On modern processors, interrupt processing is usually performed at a higher priority level than task processing.

A hardware interrupt priority solution would not accomodate the requirement to elevate task priorities above IRQ priorities.

Therefore, interrupt processing in task priority-space, also allows real-time tasks to execute at higher priority levels than interrupt handlers.

It is not possible for a thread to block when running in IRQ context.

Thread-context interrupt handlers can be preempted under the same constraints as other threads in the system.

6.2.2 Softirq Processing in Thread Context

书签:

The Softirq Daemon processing absorbs interrupt-context softirq overloads into task space, and accepts low-level transient workloads generated by various subsystems.

Interrupt response time can be improved by deferring 100% of softirq activity into softrqd. This change eliminates all latencies associated with non-preemptible interrupt-context softirq activity.

A new Kernel configuration option has been added, which allows configuring the kernel to demote softirq activity into thread context:

```
Processor type and features  --->
        [*] Thread Softirqs
```

6.2.3 Hardirq Processing in Thread Context

A top-half interrupt, or hardirq, is designed to perform the minimum amount of servicing necessary to allow the associated device to resume operation.

Multiple pending hardirqs at high frequencies can present a formidable system load, and must be prioritizable in thread context.

Thread-based IRQ processing alows hardirqs to run preemptably in thread context, and therefore significantly reduces sporadic processing delays and consequent preemption latencies within the kernel.

A new Kernel configuration option has been added, which allows configuring the kernel to demote hardirq activity into thread context:

```
Processor type and features  --->
        [*] Thread Hardirqs
```

6.3 Task-Domain Real-Time Enhancements

The primary objective in achieving effective task-level preemption latency reduction in the Linux kernel, is to reduce the total size, and number of non-preemptible critical sections.

Large portions of the code executed by softird and hardirqs accesses data protected by critical sections, and is therefore non-preemptible.

Additionally some interrupt handlers execute with interrupts disabled (SA_INTERRUPT), and are therefore also non-preemptible on the CPU.

The enhancements in the interrupt domain eliminate inpredictable interrupt-context processing, during Kernel operation, however they cannot eliminate non-preemptible Kernel operation.

- No significant improvements in task preemption latency can be achieved by simply moving softirq interrupt activity to task space
- No significant improvements in task preemption latency can be achieved by moving hardirq interrupt handling into task space

The design principles of the fully preemptible Real-Time Linux kernel are based on extensions to processing and locking concepts already found in the SMP Linux kernel:

6.3.1 Critical section independence

Critical sections are identified by unique names. In SMP, multiple CPUs can concurrently traverse independent critical sections in the Kernel.

Therefore, if the multiple tasks are allowed to operate inside independent critical sections at the same time, under specific conditions.

Critical sections can be protected by an alternative locking mechanism known as a Mutex. The use of a Mutex, to protect critical sections, guarantees data integrity, while extending the operation of the spinlock in several key aspects:

1. Tasks are suspended in a priority queue, when a locked critical section is encountered. Waiting tasks are activated when the critical section is unlocked

2. The kernel associates a locked critical section with a task, by storing the task's identity in the lock, when the mutex is acquired

3. Critical sections can be protected without disabling preemption

4. Priority inheritance can be applied to tasks operating in locked critical sections

6.3.2 Spinlock Substitution

The Linux 2.6 Kernel is already preemptible while tasks are executing outside of critical sections.

Replacing non-preemptible spinlocks with Mutexes, enables task-preemption inside critical sections with real-time characteristics:

- Thread ownership is associated with every locked mutex in the RT kernel
- The RT mutex provides deadlock detection
- The RT Mutex provides priority inheritance
- All locks are assumed to be preemptible, including locks declared in driver code. This design eliminates the possibility, that an inefficient locking implementation in a driver can introduce undetected, non-preemptible latency into the system
- Only a small, managable subset of locks remain non-preemptible
- Internal timing variations caused by increased preemptibility is context-switch transparent to the executing task

6.4 Performance Impact

The mutex operations introduce additional overhead, when compared to the non-preemptible spinlock kernel implementation. The additional overhead is a necessary trade-off to accommodate preemptible task scheduling.

The performance impact from executing interrupts in task space, is perceptible in network throughput, and can be detected in benchmarks of some I/O operations.

6.5 Summary

Deferral of interrupt processing in task space, combined with substitution of the real-time mutex for spin-locks, results in the Linux kernel code becoming highly preemptible.

In the fully-preemptible (PREEMPT_RT) Real-Time Linux kernel, the scheduler can preempt threads anywhere in the kernel, even while operating inside critical sections.

The underlying concept of the PREEMPT_RT kernel is expressed by a simple rule: Only code that absolutely must be non-preemptible is allowed to be non-preemptible.

6.5.1 Real-Time Kernel Features

The fully-preemptible Linux Kernel design is based on a functional extension of the non-preemptible locking features found in the SMP Kernel. The subsequently introduced RT-Mutex data type provides the foundation of complete preemption, and provisions priority inheritance and deadlock detection.

6.5.2 SMP Foundation

The real-time problem is simplified dramatically by the robustness of the Linux SMP implementation, which already guarantees timing independence and correct operation in a concurrent environment.

The locking paradigm of the Real-Time Linux kernel is consistent with SMP-safe operation and and context-switch transparent coding standards.

6.5.3 Preemptible BKL

The BKL is implemented as a mutex the Linux RT kernel. This transition allows code which locks the BKL, to become preemptible.

7 The Real-Time Mutex

The Real-Time mutex facilitates the operation of multiple threads in multiple critical sections in a uniprocessor Linux Kernel.

7.1 Priority Inheritance

The generic Real-Time mutex is adapted for use with the Real-time kernel, and offers priority inheritance to reduce delays imparted on high-priority applications attempting to access locked critical sections.

A high priority task encountering a locked critical section must not be delayed indefinitely by a lower priority task blocking the contended critical section.

In the Real-Time Linux Kernel, multiple critical sections can be locked at the same time.

Priority Inheritance boosts the priority of a task preempted in a critical section, to the highest priority of any task waiting for the same critical section. Transitive priority inheritance allows the highest priority thread to assert its priority on multiple levels of critical section dependencies.

7.2 Deadlock Detection

Deadlocks can be created in incorrectly implemented critical section code. These deadlocks must be detected and resolved. The Linux Kernel provides Deadlock detection on the Real-Time kernel mutex.

7.3 Nested Critical Sections: Mutex and Deadlocks

Nested locking is a significant contributor to preemption latency in the Community Kernel. In the Real-Time Linux Kernel, the nesting of critical sections requires an ordering of nested locks to avoid deadlocks.

7.4 Mutex and raw-spinlock Co-existence

Not every spinlock in the Linux Kernel, can be converted to a mutex. Certain critical sections of low-level code are not preemptible, and must be protected by the legacy non-preemptible spinlock.

Examples of non-preemptible critical sections are:

- short-held locks, where a context switch would require greater overhead
- locks protecting hardware registers that must be non-preemptible for correct system operation
- locks nested within other non-preemptible spinlocks

7.4.1 Lock Nomenclature

In the RT kernel, the legacy non-preemptible spinlock implementation co-exists with the new mutex-based locks. The non-preemptible lock has been renamed to raw_spin_lock.

A new data type corresponds to the new lock name: raw_spinlock_t

The reason for creating a new name for the old lock type, lies in the architecture of the RT kernel:

The vast majority of the ten-thousand+, spinlock-protected critical sections found in the Linux kernel have been converted to blocking mutexes.

Only a small number of raw_spinlocks are retained in the RT kernel. It is much easier to change the declaration of 100 locks, than of 10,000.

Some of the locks that do remain non-preemptable, are essential for correct operation of the system. The scheduler's runqueue locks, as well as the synchronization code that synchronizes access to the real-time mutexes, are examples of non-preemptable code.

3rd-party drivers generally compile by default to be fully preemptible.

7.5 Spin-Lock Function Mapping

The initial RT Kernel Prototype had a major shortcoming in maintainability, since the patch had to explitly modify all references to operations on raw_spin_lock. This compromised the maintainability of the patch, and increased its size intolerably.

A better inplementation to accommodate the coexistence of the two distinct locking mechanisms in the RT kernel was developed by Ingo Molnar.

The function mapping operation uses a GCC preprocessor operation, that is performed on each spinlock function call, in order to map the operation to either a raw_spinlock, or a mutex.

The pre-processor operation is known as the **__builtin_types_compatible** function, which will examine the declared type of the lock and compare it to a known type-name (raw_spin_lock).

The effective function mapping is performed by the TYPE_EQUAL macro shown below:

```
#define TYPE_EQUAL(lock, type) \
     __builtin_types_compatible_p(typeof(lock),\
  type *)
```

The PICK_OP function-mapping construct is the mechanism used to allow two locking primitive to coexist without compromising the functionality of the non-RT configured kernel.

PICK_OP converts lock operations to the appropriate low-level function type (either mutex or spinlock) at compile time.

The lock operation is determined based upon the declared data type, of the operation's object.

```
#define PICK_OP(type, optype, op, lock) \
do {                                          \
     if (TYPE_EQUAL((lock), type))            \
             _raw_##optype##op((type *)(lock));\
     else if (TYPE_EQUAL(lock, spinlock_t))   \
             _spin##op((spinlock_t *)(lock)); \
     else __bad_spinlock_type();              \
} while (0)
```

The spin_lock header-file construct below, effectively maps the spin_lock function call using the PICK_OP primitive.

```
#define spin_lock(lock) \
     PICK_OP(raw_spinlock_t, spin, _lock, lock)
```

7.5.1 PICK_OP Drawbacks

The deeply embedded low-level function structure makes it difficult to determine from an arbitrary piece of code, whether the code is operating on a mutex, or on a raw_spinlock.

It is important to inspect the actual variable declaration to resolve the ambiguity.

7.6 Mutex Operation

7.6.1 NON-Preemptible ("Raw") Spinlock Operation

If the lock is declared as a raw_spinlock_t, spin_lock is mapped to _raw_spin_lock:

```
#define _raw_spin_lock(lock) \
do { \
     preempt_disable(); \
     __raw_spin_lock(lock); \
     __acquire(lock); \
} while(0)
```

The lock-operation from this point on is identical to the community kernel implementation. The __raw_spin_lock function that is called in this implementation, is the architecture-specific, atomic function that operates on the legacy, non-preemtable (raw_spinlock) data type in the SMP kernel's physical memory. In UP systems, this lock is still optimized to a preempt_disable operation. The additional _ has been added to accommodate the additional abstraction layer required for the PICK_OP macro. The developer should be aware that use of the raw_ function types and their lower level support functions discussed here, results in code that is incompatible with some Communitys Linux 2.6 kernels.

7.6.2 RT Mutex Operation

If the lock is declared as a spin_lock, the PICK_OP construct maps the function call to _spin_lock, and the operations performed will use a mutex parameter:

```
void _spin_lock(spinlock_t *spin)
{
     __spin_lock(spin, CALLER_ADDR0);
}
```

```
static void __spin_lock(spinlock_t *lock,
                        unsigned long eip)
{
     SAVE_BKL(_down_mutex(&lock->lock, eip));
}

#define SAVE_BKL(ACTION)                    \
{                                           \
     struct task_struct *task = current;    \
     unsigned int saved_lock_depth;         \
                                            \
     saved_lock_depth = task->lock_depth;   \
     task->lock_depth = -1;                 \
                                            \
     might_sleep();                         \
     ACTION;                                \
                                            \
     task->lock_depth = saved_lock_depth;   \
}

static void _down_mutex(struct rt_mutex *lock,
unsigned long eip)
{
     TRACE_WARN_ON(lock->save_state != 1);
     __down_mutex(lock, eip);
}
```

The __down_mutex operation, finally performs the locking, and allows the task to suspend itself if the mutex is contended.

```
static void __sched __down_mutex(
     struct rt_mutex *lock,
     unsigned long eip)
{
...
}
```

7.7 Locking API

Low-level locking will see continued optimization and development in the Linux kernel . The developer is encouraged to utilize the standard high-level functions (spin_lock, etc.), for all non-time-critical code development.

Non-preemptible locks must be declared explicitly, using raw_spinlock_t or DECLARE_RAW_SPINLOCK definitions.

The proper transition to the real-time mutex type is guaranteed, when using conventional high-level locking functions in code

8 Kernel Timers

8.1 Brief History of Timers

The need for some type of timer is long standing in Unix. A simple example is the *alarm* system call. The *alarm* system call will schedule a time out in an arbitrary number of second. As time has gone on timers have grow in complexity and the user space portion has been standardized by POSIX in [1]. Issue 6 IEEE Std 1003.1, 2004 Edition http://www.opengroup.org/onlinepubs/009695399

8.1.1 Linux system timers

The timer subsystem in Linux was designed mainly to function off of the the kernels notion of time called a *jiffie* . All timers would have their timeout converted to jiffies , then a Linux system timer would be initialized to handle it. The POSIX timers where added as an extension to this, so a POSIX timer created in user space would eventually trickle down to a Linux system timer.

Linux system timers are handled in by a per cpu structure of cascading vectors, shown below.

```
typedef struct tvec_s {
     struct list_head vec[TVN_SIZE];
} tvec_t;

typedef struct tvec_root_s {
     struct list_head vec[TVR_SIZE];
} tvec_root_t;

struct tvec_t_base_s {
     struct timer_base_s t_base;
     unsigned long timer_jiffies;
     tvec_root_t tv1;
     tvec_t tv2;
     tvec_t tv3;
     tvec_t tv4;
     tvec_t tv5;
} ____cacheline_aligned_in_smp;
```

When a timer is scheduled it is places into one of five vectors *tv1*, *tv2*, *tv3*, *tv4*, or *tv5*. The vector is picked based in how much time needs to pass before the timer expires. Each vector can be thought of as an array of jiffies moving into the future. All that needs to be done to insert a timer is to place it in the vector position equal to it's time out value in jiffies. The insert operation amounts to a hash table insert, and it operates in O1 or constant time. The removal of a timer is also constant time.

8.1.2 Timer Cascade

The Linux timer system requires attention at every timer interrupt. When the timer interrupt in triggered this indicates that the jiffies value will be incremented. By incrementing the jiffies value this also means that all the time vectors must be shifted to the right. To accomodate this shifting the *cascade* functions was implemented.

```
static int cascade(tvec_base_t *base,
                   tvec_t *tv,
                   int index)
{
     /* cascade all the timers from
        tv up one level */
     struct list_head *head, *curr;

     head = tv->vec + index;
     curr = head->next;
     /*
      * We are removing _all_ timers from
      * the list, so we don't have to
      * detach them individually, just
      * clear the list afterwards.
      */
     while (curr != head) {
             struct timer_list *tmp;

             tmp = list_entry(curr,
                              struct timer_list,
                              entry);
             BUG_ON(tmp->base != &base->t_base);
             curr = curr->next;
             internal_add_timer(base, tmp);
     }
     INIT_LIST_HEAD(head);
     return index;
}
```

This function then runs on all time vectors and essentially re-inserts all timers in the system at that time. The benefits to this system are seen only in it's fast timer insertion and deletion. It's important to note that, when this system was developed, most timers where inserted then deleted prior to their expiration which has been observed by George Anzinger, and Thomas Gleixner in [5].

The cascade function runs with interrupts and preemption disable. As noted by Ingo Molnar on the Linux Kernel Mailing List, he observed 16 million timers running at once.

This presents some very real latency problems. "Another source of regression is the fact that quite a lot of timer functions execute long lasting codepaths. E.g. in the networking code rt_secret_rebuild() does a loop over rt_hash_mask (1024 in my case), over entries and over some subsequent variable sized loops inside each step. On a 300MHZ PPC system this accumulated to a worst case total of >5ms [including cascade] ... " [5].

The function of the jiffie is to represent a period of time, however this period is not static. A

new feature was recently added to Linux that allows the span of a jiffie to be selected during pre-compile kernel configuration. " [This changing jiffie period causes an] increased necessity to move non-expired timers from the outer [time vectors] to the primary [time vector]" [5]. These movements are done inside the cascade function.

8.2 High-Resolution Timers

The POSIX standard discussed timers of different resolution [1]. UTIME [8] was the first project that implemented sub jiffie timer resolution in the Linux Kernel. Subsequently George Anzinger introduced the High-Resolution Timers patch (HRT), hereafter referred to as HRT.

8.2.1 High Resolution Timer Design

The design of HRT was to use a Linux system timer to organize all the sub jiffie timers. Each Linux system timer represented a queue of timers that would be triggered in that jiffie. HRT can also use it's own timer interrupt. This timer interrupt would be used to trigger each of the timers queued by HRT's Linux system timer.

The other mode of operation is for HRT to use only the system timer, however it's desirable to have an independent timer. HRT was designed for microsecond resolution , but was molded around jiffie resolution.

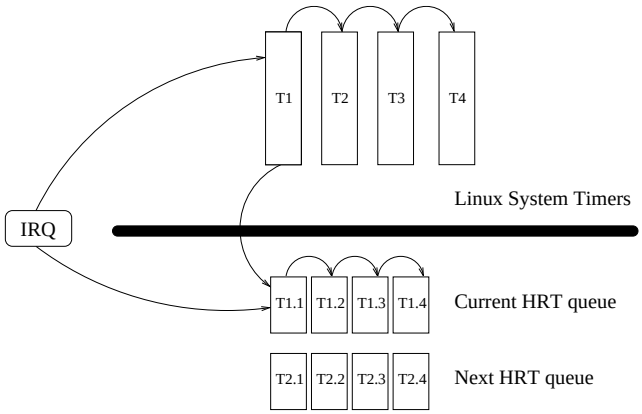

FIGURE 1: *High level HRT design*

The HRT design was extended from the design used by the UTIME project.

"To achieve microsecond resolution timing, we have added a field to the timer_list structure (see include/linux/timer.h). This field (usec) indicates the microsecond within the current "jiffy" that the timer is to timeout. To maintain compatability with the rest of the kernel, we have introduced the notion of a software clock. This clock maintains the "jiffies" granularity and calls the "do_timer()" routine every jiffy, so that the rest of the system which depends on this would run properly." [8]

The difference between UTIME and HRT, is that UTIME re-factored the Linux System Timers to have microsecond accuracy while maintaining compatibility. HRT is built on top of Linux System Timers and introduced a separate interrupt to maintain microsecond resolution.

8.3 Ktimers

Ktimers is a current project [5] by Thomas Gleixner and Ingo Molnar. Like HRT, and UTIME, Ktimers can also provides high resolution timers. "ktimers seperate[s] the "timer API" from the "timeout API". ktimers are used for ... - nanosleep - posixtimers - itimers" [5] . This separation of timers that usually do expire, from timers that usually don't (i.e. timeout timers) is fairly novel. The seperation allow a clean implementation for Posix timers, which currently use Linux System Timers.

8.4 Design of Ktimers

All timers under Ktimers are stored in a red-black tree. This provides a fast insertion however it's not constant time like Linux System Timers. Ktimers uses a structure called ktime_t to hold the expiration time. Ktimers was created so all of it's timers are scheduled ith the configured system resolution. Ktimers uses nanosecond timeouts specifications. The first implementation was based on scalar 64 bit nanosecond variables. Recently the internal representation of time values was changed to a hybrid of a modified timespec structure:

```
typedef union {
     s64     tv64;
     struct {
#ifdef __BIG_ENDIAN
     s32     sec, nsec;
#else
     s32     nsec, sec;
#endif
     } tv;
} ktime_t;
```

The data type was introduced to improve performance on 32bit CPUs by using 32 bit math operations for conversion between the Posix timespec structure and the 64-bit nanosecond value used by Ktimers. 64bit systems use the scalar nanosecond-based representation of internal time values.

Like HRT, Ktimers uses it's own interrupt to trigger timers. The difference between Ktimers and HRT is that ktimers doesn't interact with the Linux

System Timers. Ktimer time keeping is entirely independent of the jiffies value. This also allows for clean usage of the 64-bit nanosecond values. If these values where based off of jiffies like in the current Linux Posix timers code, then expiration values would have to constantly converted between time representation.

A good summary of the in-kernel API is given in: http://lwn.net/Articles/151793/ .

9 Robust User-Space Mutex

9.1 Robust Mutex Introduction

Until recently, there has been very little progress made in the area of user space real-time.

There are currently two user space projects for a robust mutex implementations. The longest running of which is FUSYN maintained by Inaky Perez-Gonzalez of Intel, and the other Robust Futex by Dave Singleton of MontaVista and Todd Kneisel of Bull.

The focus of this section is to give some overview of robust mutex projects, and then to specifically focus on Dave Singleton's and Todd Kneisel's implementation.

9.2 Real Time

Mutexes are software syncronization mechanisms that can be used to control and track access to shared resources. A Real Time mutex refers to a mutex that is aware of the priorities of it's lockers. There are several features that play into this. Priority inheritence, and priority queuing are likely the most import Real Time features applied to a mutex.

Priority Inheritence is the process of changing the priority of a mutex owner based on the highest priority task wait on the mutex. This process is transitive. So , priority inheritence operates on all mutex owners in a chain of sleeping owner and bottoms out on the running owner.

Priority Queuing referes to the ordering of the tasks waiting on a given mutex. This ordering is done from highest priority to lowest priority. It's desirable for this to be done in O(1) or a constant time data structure.

These Real Time features require that the owner of a mutex be tracked. Tracking of a mutex owner is also nessecary for **Robustness**.

9.3 Robustness

Robustness refers to the addition of a tracking mechanism for user space mutex lockers. Robust mutexes are used primarily by user space processes. The tracking is used to follow when a user space process dies while holding a mutex. In order to track when a process dies, the mutex must have some hooks into the kernel. FUTEX [4] was the first user space mutex to provide the nessesary kernel hooks.

The term Robust Mutex comes from Sun Microsystems, who was one of the first to standardize this type of extention . A key distinction to make is where a mutex is used i.e. userspace or kernel space. Robustness is simple to track inside the kernel. It's actually just considered an error condition for a kernel process to exit while holding a mutex or semaphore. However, in userspace it becomes a particular challenge.

9.4 Futex

FUTEX was the first proposal for a user space mutex implemented in the kernel. "Fast userlevel locking is an alternative locking mechanism to the typical heavy weight kernel approaches such as fcntl locking and System V semaphores. [In the FUTEX implemetation], multiple processes communicate locking state through shared memory regions and atomic operations. Kernel involvment is only nessecary when there is contention on a lock, in order to perform queuing and scheduling functions" [4]. The FUTEX implemetation makes the current assumption that most locks will be uncontended. This provides a common fast path that doesn't enter the kernel.

FUTEX was designed to be simply a fast user space mutex. It was not designed with robustness or real time applications in mind. The queuing functionality of the FUTEX implementation is not designed to directly take a waiting processes priority into account. The implementation is to "wake all waiting processes and have them recontend for the lock ... refered to as random fairness" [4]. This will allow a high priority task to take the lock first , but it doesn't mean the highest priority waiting task will take the lock.

The FUTEX designers also make specific note of the robustness problem. "... a solution needs to be found that enabled recovery of "dead" locks. We define unrecoverable locks as those that have been acquired by a process and the process terminates without releasing the lock. ... We have not addressed this problem in our prototypes yet." [4]

The FUTEX implementation today is largely the same as the original design. It stands out as the main example in linux of a fast user space mutex. This design was leveraged by Todd Kneisel to add simple robustness, then again by Dave Singleton .

9.5 Userspace Implementation

The developers of both robust mutex projects have implemented their mutexes with Posix in mind. Both projects provide some changes to Glibc. Glibc modification must be made to provide the Posix API.

The robustness extensions to a user space mutex isn't covered by Posix. This unfortunate detail requires the developers to define their own standard. This has made the user space API some what different depending on who has made the changes. All projects implement at least a subset of the original Sun Microsystems API .

```
int pthread_mutexattr_setrobust_np
        (pthread_mutexattr_t *attr,
         int *robustness);
```

With one of the following attributes sent in the robustness field.

PTHREAD_MUTEX_ROBUST_NP signals that this mutex should be robust. If the owner dies, the next waiter will be woken up and return with the error flag EOWNWERDEAD.

PTHREAD_MUTEX_NOROBUST_NP signals that robustness should not apply to this mutex.

Although atleast these two flags are available in all implementation, there are likely more depending on which implementation is used.

9.6 Fusyn

FUSYN maintained by Inaky Perez-Gonzalez of Intel has been around since 2003. Fusyn started out as the rtfutex, "Once the requirements were laid out, we first tried modifying the futex code in kernel/futex.c adding functionality while maintaining the original futex interface" [6]. This approach would have been similar to what Dave Singleton and Todd Kneisel created except that the rtfutex would have had to embody a complete real time mutex.

The rtfutex approach was eventually abandoned. "This design *andit'simplementation* was broken: the futexes are designed to be queues, and they cannot be stretched to become mutexes-it is simply not the same. The result was a bloated implementation" [6]. By not working inside the kernel Fusyn was able to dictate the structure of their real time mutex, and the system calls used to access it.

Fusyn ended up being comprised of four main pieces fuqueues, fulocks, vlocators, and vfulocks [6]. Of these pieces the fulock was the base level kernel mutex, and vfulocks is the userspace glue.

Fusyn is still an active project , however it has never been accepted into a mainline kernel. The reason for this is because it created a new blocking primitive which the kernel already has in semaphores. The fulock doesn't replace the semaphores, and has no in kernel users.

9.7 Dave Singleton and Todd Kneisel's work

In late 2004 the Preempt Real-Time kernel changes started. These changes introduced an RT-Mutex, which included features like **Priority Inheritence** and **Priority Queuing**. The RT-Mutex was also created to be architecture independent, in contrast with Linux system semaphores which are architecture dependent. However, the RT-Mutex was not being exported to user space processes which made it strictly an in kernel API . The RT-Mutex is used to replace a very high percentage of system semaphores users, something that fulocks is lacking.

In June 2005 Todd Kneisel released code that modified the current Linux FUTEX implementation to provide robustness [7]. These changes introduced the pthread API calls used by Sun Microsystem for robustness. In order to accomplish robustness Todd Kneisel first had to implement mutex ownership tracking. As stated earlier, mutex ownership tracking is necessary for all real time feature.

Dave Singleton originally started working on integration of the RT-Mutex into FUSYN to replace the fulock. This turn out the be a very complex. The fulock appeared to be made specifically for use in FUSYN. It did not seem straight forward to replace it. Eventually and idea surfaced to integrate the RT-Mutex into the FUTEX code incorporating the work from Todd Kneisel as a base.

The marriage between the RT-Mutex and FUTEX turned out the be a much more tractable solution. However, this integration has not been around nearly as long as FUSYN, so it will likely continue to be in flux. It also may end up incorporating work from FUSYN .

10 Summary

Real time progress inside the kernel has been moving forward at an incredible pace.

Spin locks have been converted to mutexes, there are thread-context hard irqs, threaded soft irqs, many interrupt off section have been removed, in addition to countless other changes.

The preemption latency of the Linux Kernel has been reduced by one, to several orders of magnitude, allowing it to compete head-to-head with subkernel solutions for Linux, as well as commercial real-time operating systems.

Recently, Ktimers have been introduced into the real-time patch, allowing real-time tasks to interact with very low latency to fine-granular high-performance timer facilities that allow implementation of real-time control loops achieving unprecedented precision.

Finally, the user-space real-time frontier is giving way, with a foray into exposing the RT mutex to userspace via a standard Kernel API. Userspace tasks are able to leverage efficient priority inheritance, deadlock detection, and robust dead-owner lock recovery protocols.

Its more than enough to make your head spin.

We wish to extend our gratitude and special thanks for patience, understanding, and unabated enthusiam to:

- Dave Singleton
- Doug Niehaus
- Inaky Perez-Gonzalez
- Ingo Molnar
- Kevin Morgan
- Mark Orvek
- Steve Rostedt
- Thomas Gleixner
- Paul McKenney

and many many others who have contributed to the Real-Time preemption project.

References

[1] Ieee std 1003.1, 2004. http://www.opengroup.org/onlinepubs/009695399/toc.htm.

[2] Paul Barton-Davis. a joint letter on low latency and linux, Jun 2000. http://marc.theaimsgroup.com/?l=linux-kernel&m=96223850230609&q=raw.

[3] Kevin Dankwardt. Real-time and linux, part 2: the preemptible kernel, March 2001. http://www.linuxjournal.com/article/5742.

[4] H. Franke, M. Kirkwood, and R. Russell. Fuss, futexes and furwocks: Fast userlevel locking in linux. pages 479–495, 2002. http://www.linux.org.uk/ajh/ols2002_proceedings.pdf.gz.

[5] T. Gleixner. [announce] ktimers subsystem, Sep 2005. http://lkml.org/lkml/2005/9/19/124.

[6] I. P. Gonzales, S. Searty, A. Li, D. P. Howell, and B. Hu. I would hate user space locking if it weren't that sexy... volume 2, pages 403–423, 2004. http://www.linuxsymposium.org/proceedings/LinuxSymposium2004_V2.pdf.

[7] T. Kneisel. robust futexes for 2.6.12-rc6, June 2005. http://lkml.org/lkml/2005/6/15/228.

[8] D. Niehaus, R. Menon, S. Balaji, F. Ansari, J. Keimig, and A. Sheth. Microsecond resolution timers for linux, 1997. http://www.ittc.ku.edu/utime/.