

Priority Boosting Preemptible RCU

Paul E. McKenney
Linux Technology Center
IBM Beaverton
paulmck@linux.vnet.ibm.com
paul.mckenney@linaro.org

ABSTRACT

Read-copy update (RCU) is a light-weight synchronization mechanism that has been used in production for well over a decade, most recently, as part of the Linux kernel. The key concept behind RCU is the ability of RCU update-side primitives (`synchronize_rcu()` and `call_rcu()`) to wait on pre-existing RCU read-side critical sections, which are delimited by `rcu_read_lock()` and `rcu_read_unlock()`. The time required for all pre-existing RCU read-side critical sections to complete is termed an RCU grace period. A common usage pattern is to remove an element from a data structure, wait for an RCU grace period to elapse, then free that element.

Older implementations of RCU operated by suppressing preemption across the RCU read-side critical sections, but more recent implementations designed for real-time use permit such preemption. This can lead to a priority-inversion problem, where a low-priority non-real-time task is preempted within an RCU read-side critical section by several medium-priority real-time tasks (at least one per CPU). This situation prevents any subsequent RCU grace periods from completing, which prevents the corresponding memory from being freed, which can exhaust memory, which can block a high-priority real-time task that is attempting to allocate memory.

This situation is similar to lock-based priority inversion, and, as with lock-based priority inversion, and can be solved by temporarily boosting the priority of the low-priority task that was blocked in an RCU read-side critical section.

1. INTRODUCTION

RCU is a synchronization mechanism that allows execution to be deferred until all potentially conflicting operations have completed, which greatly simplifies the design and implementation of concurrent algorithms [5]. Although RCU antecedents date back to 1980 [4], RCU attained widespread use only after its acceptance into the Linux kernel in 2002, where it has since become quite heavily used, as shown in Figure 1. RCU’s popularity stems from its solution to the “existence problem” [2]. The potentially conflicting operations, termed *RCU read-side critical sections*, are bracketed with `rcu_read_lock()` and `rcu_read_unlock()` primitives. Production-quality implementations of these primitives scale linearly, are wait-free, are immune to both deadlock and livelock, and incur extremely low overheads. In fact, in server-class (`CONFIG_PREEMPT=n`) Linux-kernel builds, these two primitives incur exactly zero

overhead [3].¹

Any statement that is not within an RCU read-side critical section is a *quiescent state*, and a quiescent state that persists for a significant time period is called an *extended quiescent state*. Any time period during which each thread has occupied at least one quiescent state is a *grace period*. The `synchronize_rcu()` primitive waits for a grace period to elapse. Updates that cannot block may use an asynchronous primitive named `call_rcu()`, which causes a specified function to be invoked with a specified argument at the end of a subsequent grace period. In some (but not all) production-quality implementations, `call_rcu()` simply appends a callback to a per-thread list, and is therefore wait-free [1, 8]. Nevertheless, RCU updates do incur some overhead, so that RCU is best-suited for read-mostly situations.

Taken together, these four primitives implement RCU’s grace-period guarantee: a given grace period is guaranteed to extend past the end of any pre-existing RCU read-side critical section [9]. It is important to note that RCU provides this guarantee regardless of the memory model of the underlying computer system.

On weakly ordered systems (a category including all commodity microprocessors), RCU also provides a publish-subscribe guarantee via the `rcu_assign_pointer()` publication and `rcu_dereference()` subscription primitives. These primitives disable any compiler and CPU optimizations that might otherwise result in an RCU reader seeing a pre-initialized view of a newly published data structure. Both of these primitives have $O(1)$ computational complexity with small constant, and incur zero overhead on sequentially consistent computer systems, where “system” includes the compiler.

Although older implementations of RCU relied on disabling preemption across all RCU read-side critical sections, more recent implementations have permitted these critical sections to be preempted in order to improve real-time scheduling latency [3, 6, 7]. As noted earlier, such preemption opens the door to a priority-inversion situation where a low-priority task is preempted in an RCU read-side critical section by medium-priority real-time tasks, preventing any subsequent RCU grace periods from ever completing. If grace periods never complete, the corresponding memory is never freed, eventually running the system out of memory. The resulting hang can be expected to block even the highest-priority real-time tasks.

@@@ Roadmap

¹Sequent’s DYNIX/ptx operating system also provided zero-overhead RCU read-side primitives [8].

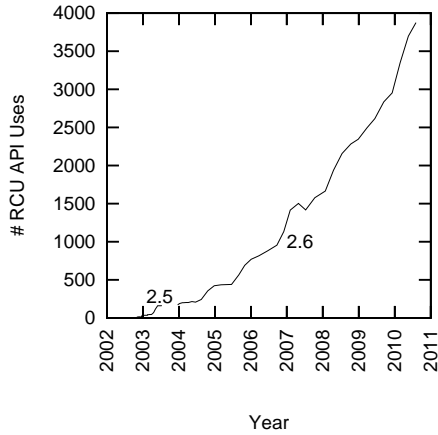


Figure 1: RCU API Usage in the Linux Kernel

2. CONTROL OF BOOSTING

The operation of RCU priority boosting is controlled by the following:

1. The new `BOOST_RCU` kernel configuration parameter, which depends on `RT_MUTEXES` (default off, at least initially). This dependency is an implementation constraint rather than a policy decision, as in absence of `RT_MUTEXES` the priority-boosting scheduler infrastructure is not compiled into the kernel. Therefore, in absence of `RT_MUTEXES`, `BOOST_RCU` simply cannot do its job. There is also a dependency on `PREEMPT_RCU` by design, given that there is no reason to boost priority for a non-preemptible RCU implementation. The default value of `BOOST_RCU` will be “n” in order to avoid an automatic Linus rejection. If experience indicates that enabling `BOOST_RCU` by default is wise, that change will be made in a later release of the kernel.
2. The priority that the `RCU_SOFTIRQ` task runs at, but only in the `-rt` patchset, at least until `PREEMPT_SOFTIRQS` reaches mainline. In kernels lacking `PREEMPT_SOFTIRQS`, the priority instead defaults to the least-important real-time priority.
3. The new `BOOST_RCU_PRIO` kernel configuration parameter, which depends on `BOOST_RCU`. This specifies the default priority to which blocked RCU readers are to be boosted. A value of zero specifies no boosting. If the value is `-1`, then the default is taken from the `RCU_SOFTIRQ` priority above. `BOOST_RCU_PRIO` depends on `BOOST_RCU`.
4. The new `rcu_boost_prio` module parameter in `kernel/rcupdate.c`, which also controls the priority to which

RCU_SOFTIRQ Priority	RCU_BOOST_PRIO Kernel Parameter	rcu_boost_prio Module Parameter	Priority
A	-1	-1	Priority A from <code>RCU_SOFTIRQ</code> priority
A	-1	C	Priority C from <code>rcu_boost_prio</code>
A	B	-1	Priority B from <code>RCU_BOOST_PRIO</code>
A	B	C	Priority C from <code>rcu_boost_prio</code>
X	-1	-1	RT Priority 1 by default
X	-1	C	Priority C from <code>rcu_boost_prio</code>
X	B	-1	Priority B from <code>RCU_BOOST_PRIO</code>
X	B	C	Priority C from <code>rcu_boost_prio</code>

Table 1: Relationship of Priority Defaults

blocked RCU readers are to be boosted. A value of zero specifies no boosting. If the value is `-1`, then the default is taken from the `BOOST_RCU_PRIO` kernel parameter above. If the value is neither zero nor a valid real-time scheduler priority, then it is treated as if it was `-1`, though a warning will be printed in this case. This parameter may also be controlled at runtime via `sysfs`.

5. The new `BOOST_RCU_DELAY` kernel parameter, which specifies the number of jiffies to wait after a given grace period begins before doing RCU priority boosting for blocked tasks that are stalling that grace period. A value of `-1` says never to do RCU priority boosting (but this may be overridden at boot time and at run time). `BOOST_RCU_DELAY` depends on `BOOST_RCU`.
6. The new `rcu_boost_delay` module parameter in `kernel/rcupdate.c`, which also controls the RCU boost delay. A value of `-1` says to take the default from the `BOOST_RCU_DELAY` kernel parameter, though any other negative value will have the same effect (but possibly accompanied by a warning). This parameter may also be controlled at runtime via `sysfs`.

The relationship between the `RCU_SOFTIRQ` priority, the `RCU_BOOST_PRIO` kernel parameter, and the `rcu_boost_prio` module parameter is involved, so their relationship is shown by Table 1. A letter A, B, or C in the first column indicates some real-time priority (which currently ranges from 1 to 99 in the Linux kernel), while the letter X indicates a kernel without threaded softirqs.² The “Priority” column indicates what controls the resulting task priority.

²As of this writing, mainline Linux does not permit threaded softirqs, aside from the non-realtime `run_ksoftirqd()` task that is used to handle overflow from the softirq environment. However, `PREEMPT_RT` kernels that include the `-rt` patchset provide a `PREEMPT_SOFTIRQS` kernel parameter that causes softirqs to be executed in the context of a real-time thread whose priority may be controlled at run time.

3. NEW DATA STRUCTURES

Although RCU priority boosting does not introduce any new data structures to RCU, it does add fields and values to a number of the existing data structures under `BOOST_RCU` ifdef as follows:

1. Add an `->rcu_prio` field to the `task_struct` structure, parallel to the existing `prio`, `static_prio`, and `normal_prio` fields. This new `->rcu_prio` field indicates the priority that RCU needs the task boosted to. It is initialized to `MAX_PRIO-1`, which is the least important possible priority.
2. Add a `RCU_READ_UNLOCK_BOOSTED` value to those values that the `->rcu_read_unlock_special` `task_struct` field.
3. Add an `rcu_boost_prio` global variable to `kernel/rcupdate.c`, which is initialized to the `BOOST_RCU_PRIO` kernel configuration parameter. This variable is exported as a module parameter and via `sysfs`, as noted in Section 2.
4. Add an `rcu_boost_delay` global variable to `kernel/rcupdate.c`, which is initialized to the `BOOST_RCU_DELAY` kernel configuration parameter. This variable is exported as a module parameter and via `sysfs`, as noted in Section 2.

The `TINY_PREEMPT_RCU` implementation additionally needs the following additional fields in existing data structures, again under `BOOST_RCU` ifdef:

1. Add a `->boost_tasks` pointer to the `rcu_preempt_ctrlblk` structure, which is initialized to `NULL`. This field points to the first tasks structure in the `blkd_tasks` lists that has needs to be boosted but has not yet been boosted, or `NULL` if there are no tasks in need of RCU priority boosting.
2. Add an `->rcu_prio` field to the `rcu_preempt_ctrlblk` structure, which is initialized to `MAX_PRIO-1`. This field records the priority to which tasks should be boosted. Note that it is possible for this field to have a different value than the `rcu_boost_prio` global variable. This situation indicates that the desired boost priority changed recently, and that all tasks that have already been boosted need to be boosted again (or deboosted in the case where the `rcu_boost_prio` global variable has changed to a lower priority.
3. Add a `->boosttime` field to the `rcu_preempt_ctrlblk` structure, which is initialized to `jiffies+rcu_boost_delay`. This field records the jiffies value at which boosting should begin.
4. Add a `->boost_task` field to the `rcu_preempt_ctrlblk` structure, which is initialized to `NULL`. This field contains a pointer to the booster kernel thread's task structure, which is needed to allow the booster kernel thread's priority to be boosted.
5. Add a `->needboost` field to the `rcu_preempt_ctrlblk` structure, which is initialized to zero. This flag signals the booster kernel thread to wake up.

6. Add a `->boostwq` field to the `rcu_preempt_ctrlblk` structure, which is initialized using the `DECLARE_WAIT_QUEUE_HEAD()` macro. This wait queue is where the booster kernel thread blocks when there is no boosting to be done.

The `TREE_PREEMPT_RCU` implementation additionally needs the same additional fields that `TINY_PREEMPT_RCU` does, but instead in each `rcu_node` structure. In addition, the `->needboost` field takes on an additional value to tell the corresponding booster kernel thread to stop.

In `PREEMPT_SOFTIRQS` kernels, additional data will be needed if the `RCU_SOFTIRQ` tasks are also to be boosted. However, the initial implementation will assume that the initial priority chosen for the `RCU_SOFTIRQ` tasks is sufficient, and will therefore refrain from boosting them.

Once the SRCU implementation is folded into `TREE_PREEMPT_RCU` implementation, an addition field will be needed to link together the corresponding `rcu_node` structures for `TREE_PREEMPT_RCU` and for the SRCU instances that are subject to priority boosting.³

4. LOCKING DESIGN

The locking design of RCU priority boosting must respect RCU's current locking design, which uses irq-disabled spinlocks, but must also accommodate the scheduler's priority-boosting locking design, which requires that boosting be undertaken with preemption enabled. The reason that preemption must be enabled while boosting is that disabling preemption would result in excessive scheduling latencies. The conflict between these two locking designs is resolved by doing bookkeeping under the irq-disabled RCU spinlocks, including adjusting the `->rcu_prio` field in the `task_struct` structure, releasing those locks, enabling interrupts, and only then invoking the scheduler to adjust the priority of the task in question as well as any processes connected to it via priority-inheritance chains.

The scheduler functions are `rt_mutex_adjust_prio()`, which adjusts the priority of the specified task, and `rt_mutex_adjust_pi()`, which adjusts the priority of any tasks in the specified task's priority-inheritance chain. Although `rt_mutex_adjust_prio()` may be invoked with interrupts disabled, `rt_mutex_adjust_pi()` must be called with not just interrupts enabled, but also with preemption enabled—otherwise, scheduling latency will be degraded. Because these scheduler primitives are idempotent, races with other priority adjustments on the same task can be rendered harmless as follows:

1. Ensure that RCU has its own priority field in the `task_struct` structure, namely `->rcu_prio`.
2. Boost only blocked tasks, so that the lock guarding the `blkd_tasks` list may be used to protect the `task_struct` structure's `->rcu_prio` field.

³At that point, SRCU instances for which SRCU read-side critical sections only acquire mutexes, but do no other general-blocking operation, can use priority boosting. In contrast, SRCU instances for which SRCU read-side critical sections do things like block waiting for network input cannot use priority boosting. After all, how are you going to decide what to boost to make the network packet arrive more quickly?

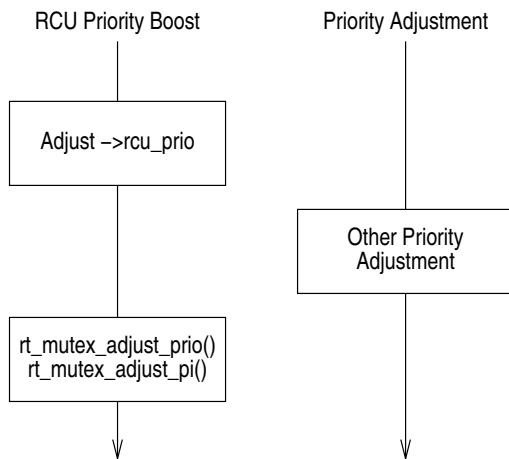


Figure 2: RCU Priority Boosting Races are Harmless

3. Modify the scheduler's functions to take the value of `->rcu_prio` into account. Not that these functions need not acquire the RCU locks: instead, they rely on gcc's guarantee that access to an aligned machine word will be done in one instruction, rather than byte-by-byte or some such. If the scheduler races with a change to `->rcu_prio`, it will see either the old value or the new value, but not some bitwise merge of both values.
4. Provide a separate kernel thread to do the boosting. This thread first modifies the relevant `->rcu_prio` field with interrupts disabled and under protection of the appropriate RCU spinlock, then invokes the scheduler function that adjusts priorities.

The overall effect is shown in Figure 2. As shown on the left, the RCU priority booster kernel thread adjusts the `->rcu_prio` field, and later invokes `rt_mutex_adjust_prio()` and `rt_mutex_adjust_pi()`. As shown on the right, between these two actions, some other thread adjusts the boosted task's priority. But because the `->rcu_prio` field has already been set, the other thread's priority adjustment takes the RCU priority boosting into account. The later actions by the RCU priority booster kernel thread are therefore redundant (the priority has already been set), but harmless.

5. OVERVIEW OF CODE CHANGES

@@@ roadmap @@@

5.1 Initialization

The following initialization is required in addition to that called out in Section 3:

1. In the `INIT_TASK()` macro, set `->rcu_prio` to `MAX_PRIO-1`.
2. In the `sched_fork` function, set `->rcu_prio` to `MAX_PRIO-1`.

5.2 Priority Booster Kernel Thread

The priority booster kernel threads cannot be created until the scheduler is up and running, and therefore cannot be

created in `rcu_init()`.⁴ Instead, these can be started via `kthread_run()` from `rcu_scheduler_starting()`, which currently enables debug checks that are disabled during early boot, and disables single-CPU optimizations that operate only during early boot. This means that the `ifdefs` covering the definitions of `rcu_scheduler_active` and `rcu_scheduler_starting()` in `TINY_PREEMPT_RCU` must now include `RCU_BOOST`. The return value from `kthread_run()` is a pointer to the `task_struct` structure, which must be recorded in the appropriate `->boost_task` field.

Additional startup/shutdown work is required for `TREE_PREEMPT_RCU`:

1. Each booster thread must be affinity'd to the set of CPUs associated with the corresponding `rcu_node` structure.
2. When the last CPU associated with the corresponding `rcu_node` structure goes offline, the corresponding booster thread must be stopped via `kthread_stop()` as follows:
 - (a) Set `->needboost` to two to indicate a need to stop.
 - (b) Wake up the booster kernel thread.
 - (c) Invoke `kthread_stop()`.

Note that this process must be carried out *after* all blocked tasks have been migrated to the root `rcu_node` structure. One way to accomplish this is to place the new code near the end of the `rcu_preempt_offline_tasks()` function.

3. When the first CPU associated with the corresponding `rcu_node` structure comes online, the corresponding booster thread must be created via `kthread_start()`. The `rcu_preempt_init_percpu_data()` function is a good home for this functionality, but only for the CPU-online case. An explicit check for `rcu_scheduler_active` is required to avoid invoking `kthread_start()` before the scheduler is ready. Carrying this out before the call to `rcu_init_percpu_data()` allows a simple test of `->qsmaskinit` being equal to zero.

Once created, the booster kernel thread operates as follows:

1. Blocks on the combination of `->needboost` and `->boostwq` using `wait_event()` so as to wake up when `->needboost` is set to one.
2. If the value of `->needboost` is two, invoke `kthread_stop()` to terminate execution.
3. Disable interrupts, and, if in `TREE_PREEMPT_RCU`, acquire the `rcu_node` structure's `->lock`.
4. Enter an RCU read-side critical section via `rcu_read_lock()`.
5. Check the `->boost_tasks` pointer. If it is `NULL`, set `->needboost` to zero, release the `->lock` (if in `TREE_PREEMPT_RCU`), re-enable interrupts, and restart from the beginning. Otherwise, continue.

⁴I know, because I have tried!

6. Set local variable `p` to the value of `->boost_tasks`, but translated back to the `task_struct` structure.
7. Advance `->boost_tasks` to the next element of the `->blkd_tasks` list. If there is no next element, instead set `->boost_tasks` to `NULL` and set `->needboost` to zero.
8. (Scheduling-latency considerations may require periodic clearing of `->needboost` under the lock.)
9. Set `p->rcu_prio` to `->rcu_prio`.
10. Set the `RCU_READ_UNLOCK_BOOSTED` bit in the `p->rcu_read_unlock_special` bitmask.
11. If in `TREE_PREEMPT_RCU`, release `->lock` and in either case re-enable interrupts. This has the side-effect of re-enabling preemption.
12. Invoke `rt_mutex_adjust_prio(p)` and `rt_mutex_adjust_pi(p)`.
13. Exit the RCU read-side critical section via `rcu_read_unlock()`.
14. Restart from the beginning.

In the `TREE_PREEMPT_RCU` case in `PREEMPT_SOFTIRQS` kernels, the booster thread must also control the priority of each of the `RCU_SOFTIRQ` tasks associated with CPUs corresponding to that booster thread's `rcu_node` structure. However, the booster thread need not de-boost the `PREEMPT_SOFTIRQS` tasks below their original priority.

5.3 De-Boosting

The de-boosting process is much simpler than the boosting process described in Section 5.2 because:

1. The task is operating on itself, and therefore need not enter an RCU read-side critical section.
2. The task is running, and therefore must have an empty priority-inheritance list. It is therefore unnecessary to invoke `rt_mutex_adjust_pi()`, which in turn makes it unnecessary to ensure that preemption is enabled.

The de-boosting takes place in `rcu_read_unlock_special()` with interrupts enabled and, in the case of `TREE_PREEMPT_RCU`, with the `rcu_node` structure's `->lock` already held. The following procedure therefore suffices:

1. Set `current->rcu_prio` to `MAX_PRIO-1`.
2. Invoke `rt_mutex_adjust_prio(current)`.

5.4 Scheduling-Clock Interrupt

TBD.

5.5 Statistics

TBD.

5.6 Changes to Scheduler Code

TBD `rcu_adjust_prio()` and changes to `rt_mutex_getprio()`.

6. TESTING

Testing will be carried out by the existing `rcutorture` module in the Linux kernel. This module will be modified as follows:

1. Add a `test_boost` module parameter, which defaults to one. A value of one says to test RCU priority boosting only if the specified flavor of RCU supports this notion, while a value of two says to test RCU priority boosting even if the specified flavor of RCU does not support this notion. This latter is useful for testing the test.
2. Add a `can_boost` flag to the `rcu_torture_ops` structure.
3. If the values of `test_boost` and `can_boost` indicate that boosting should be tested, a high-priority real-time task is spawned, one per CPU. These tasks periodically run in unison, periodically registering callbacks and checking for their completion. The time that each thread should wait is controlled by the new `rcu_boost_delay_jiffies()` function – if a given grace period does not complete in twice that value plus (say) ten jiffies, the thread complains that RCU priority boosting is not working.

@@@

7. CONCLUSIONS

@@@

Legal Statement

This work represents the views of the author and does not necessarily represent the view of IBM.

Linux is a copyright of Linus Torvalds.

Other company, product, and service names may be trademarks or service marks of others.

8. REFERENCES

- [1] DESNOYERS, M., MCKENNEY, P. E., STERN, A., DAGENAIS, M. R., AND WALPOLE, J. User-level implementations of read-copy update. Submitted to IEEE TPDS, December 2009.
- [2] GAMSA, B., KRIEGER, O., APPAVOO, J., AND STUMM, M. Tornado: Maximizing locality and concurrency in a shared memory multiprocessor operating system. In *Proceedings of the 3rd Symposium on Operating System Design and Implementation* (New Orleans, LA, February 1999), pp. 87–100. Available: http://www.usenix.org/events/osdi99/full_papers/gamsa/gamsa.pdf [Viewed August 30, 2006].
- [3] GUNIGUNTALA, D., MCKENNEY, P. E., TRIPLETT, J., AND WALPOLE, J. The read-copy-update mechanism for supporting real-time applications on shared-memory multiprocessor systems with Linux. *IBM Systems Journal* 47, 2 (May 2008), 221–236. Available: <http://www.research.ibm.com/journal/sj/472/guniguntala.pdf> [Viewed April 24, 2008].
- [4] KUNG, H. T., AND LEHMAN, Q. Concurrent maintenance of binary search trees. *ACM Transactions on Database Systems* 5, 3 (September 1980), 354–382. Available: <http://portal.acm.org/citation.cfm?id=320619&d1=GUIDE>, [Viewed December 3, 2007].

- [5] MASSALIN, H. *Synthesis: An Efficient Implementation of Fundamental Operating System Services*. PhD thesis, Columbia University, New York, NY, 1992.
- [6] MCKENNEY, P. E., AND SARMA, D. Towards hard realtime response from the Linux kernel on SMP hardware. In *linux.conf.au 2005* (Canberra, Australia, April 2005). Available: <http://www.rdrop.com/users/paulmck/RCU/realtimeRCU.2005.04.23a.pdf> [Viewed May 13, 2005].
- [7] MCKENNEY, P. E., SARMA, D., MOLNAR, I., AND BHATTACHARYA, S. Extending RCU for realtime and embedded workloads. In *Ottawa Linux Symposium* (July 2006), pp. v2 123–138. Available: http://www.linuxsymposium.org/2006/view_abstract.php?content_key=184 <http://www.rdrop.com/users/paulmck/RCU/OLSrtRCU.2006.08.11a.pdf> [Viewed January 1, 2007].
- [8] MCKENNEY, P. E., AND SLINGWINE, J. D. Read-copy update: Using execution history to solve concurrency problems. In *Parallel and Distributed Computing and Systems* (Las Vegas, NV, October 1998), pp. 509–518. Available: <http://www.rdrop.com/users/paulmck/RCU/rclockpdcproof.pdf> [Viewed December 3, 2007].
- [9] MCKENNEY, P. E., AND WALPOLE, J. What is RCU, fundamentally? Available: <http://lwn.net/Articles/262464/> [Viewed December 27, 2007], December 2007.