Locking in OS Kernels for SMP Systems

From the seminar
Hot Topics in Operating Systems

TU Berlin, March 2006

Arwed Starke

Abstract: When designing an operating system kernel for a shared memory symmetric multiprocessor system, shared data has to be protected from concurrent access. Critical issues in this area are the increasing code complexity as well as the performance and scalability of a SMP kernel. An introduction to SMP-safe locking primitives, and how locking can be applied to SMP kernels will be given, and we will focus on how to increase scalability by reducing lock contention, and the growing negative impact on locking performance by caches and memory barriers. New, performance-aware approaches for mutual exclusion in SMP systems will be presented, that made it into today's Linux 2.6 kernel: The SeqLock and the read-copy-update (RCU) mechanism.

1 Introduction

1.1 Introduction to SMP systems

As Moore's law is about to fail, since clock speeds can not be raised by a factor of two every year any more as it used to be in the "good old times", most of the gains in computing power are now achieved by increasing the number of processors or processing units working parallel: The triumph of SMP systems is inevitable.

The abbreviation SMP stands for tightly coupled, shared memory symmetric multiprocessor system. A set of equal CPUs accesses a common physical memory (and I/O ports) via a shared front side bus. Thus, the FSB becomes a contended resource. A bus master manages all read/write accesses to the bus. A read or write operation is guaranteed to complete *atomically*, which means, before any other read or write operation is carried out *on the bus*. If two CPUs access the bus within the same clock cycle, the bus master nondeterministically (from the programmers view) selects one of them to be first to access the bus. If a CPU accesses the bus while it is still occupied, the operation is delayed. This can be seen as a hardware measure of synchronisation.

1.2 Introduction to Locking

If more than one process can access data at the same time, as is the case in preemptive multitasking systems and SMP systems, mutual exclusion must be introduced to protect this shared data.

We can divide mutual exclusion into three classes: Short-term mutual exclusion, short-term mutual exclusion with interrupts, and long-term mutual exclusion [Sch94]. Let us take a look at the typical uniprocessor (UP) kernel solutions for these problem classes, and why they do not work for SMP systems.

Short-term mutual exclusion refers to preventing race conditions in short critical sections. They occur, when two processes access the same data structure in memory "at the same time", thus causing inconsistent states of data. On UP systems, this could only occur if one process is preempted by the other. To protect critical sections, they are guarded with some sort of `preempt_disable`/`preempt_enable` call to disable preemption, so a process can finish the critical section without being interrupted by another process. In a non-preemptive kernel, no measures have to be taken at all. Unfortunately, this does not work for SMP systems, because processes do not have to be preempted to run "parallel", there can be two processes executing the exact same line of code at the exact same time. No disabling of preemption will prevent that.

Short-term mutual exclusion with interrupts involves interrupt handlers that access shared data. To prevent interrupt handler code from interrupting a process in a critical section, it is sufficient to guard a critical section in the process context with some sort of `cli`/`sti` (disable/ enable all interrupts) call. Unfortunately, this approach does not work on SMP systems as well, because all other CPUs' interrupts are still active and can execute the interrupt handler code at any time.

Long-term mutual exclusion refers to processes being held up accessing a shared resource for a longer time. For example, once a write system call to a regular file begins, it is guaranteed by the operating system that any other read or write system calls to the same file

will be held until the current one completes. A write system call may require one or more disk I/O operations in order to complete the system call. Disk I/O operations are relatively long operations when compared to the amount of work that the CPU can accomplish during that time. It would therefore be highly undesirable to inhibit preemption for such long operations, because the CPU would sit idle waiting for the I/O to complete. To avoid this, the process executing the write system call needs to allow itself to be preempted so other processes can run. As you probably already know, semaphores are used to solve this problem. This also holds true for SMP systems.

2 The basic SMP locking primitives

When we talk about mutual exclusion, we mean that we want changes to appear as if they were an atomic operation. If we can *not* update data with an atomic operation, we need to make an update uninterruptible and sequentialize it with all other processes that could access the data. But sometimes, we *can*.

2.1 Atomic Operations

Most SMP architectures possess some operations that read and change data within a single, uninterruptible step, called atomic operations. Common atomic operations are test and set (TSR), which returns the current value of a memory location and replaces it with a given new value, compare and swap (CAS), which compares the content of a memory location with a given value, and, if they equal, replaces it with a given new value, or the load link/ store conditional instruction pair (LL/SC). Many SMP systems also feature atomic arithmetical operations: Addition by given value, subtraction by given value, atomic increment, decrement, among others.

The table below is an example of how the line `counter++` might appear in assembler code ([Sch94]). If this line is executed at the same time by two CPUs, the result is wrong, because the operation is not atomical.

	CPU 1			**CPU 2**		
Time	Instruction Executed		Register R0	**Value of Counter**	Instruction Executed	Register R0
1	load	R0, counter	0	0	load R0, counter	0
2	add	R0, 1	1	0	add R0, 1	1
3	store	R0, counter	1	1	store R0, counter	1

To solve such problems without extra locking, one can use an atomic increment operation as shown in Listing 1. (In Linux, the atomic operations are defined in atomic.h. Operations not supported by the hardware are emulated with critical sections.)

```
atomic_t counter = ATOMIC_INIT(0);
atomic_inc(&counter);
```

Listing 1: Atomic increment in Linux.

The shared data is still there, but the critical section could be eliminated. Atomic *updates*

can be done on several common occasions, for example the replacement of a linked list element (Listing 2). Not even a special atomic operation is necessary to do that.

```
// set up new element
new->data = some_data;
new->next = old->next;
// replace old element with it
prev->next = new;
```

Listing 2: Atomical update of single linked list, replace "old" with "new"

Non-blocking synchronisation algorithms solely rely on atomic operations.

2.2 Spin Locks

Spin locks are based on some atomic operation, for example test and set. The principle is simple: A flag variable indicates if a process is currently in the critical section (lock_var = 1) or if no process is in the critical section (lock_var = 0). A process spins (busy waits) until the lock is reset, then sets the lock. Testing and setting of the lock status flag must be done in one step, with an atomic operation. To release the lock, a process resets the lock variable. Listing 3 shows a possible implementation for `lock` and `unlock`.

```
void lock(volatile int *lock_var_p) {
      while (test_and_set_bit(0, lock_var_p) == 1);
}

void unlock(volatile int *lock_var_p) {
      *lock_var_p = 0;
}
```

Listing 3: Spin lock [Sch94]

Note that a spin lock can not be acquired recursively – it would deadlock on the second call to `lock`. This has two consequences: A process holding a spin lock may not be preempted, or else a deadlock situation could occur. And spin locks can not be used within interrupt handlers, because if an interrupt handler tries to acquire a lock that is already held by the process it interrupted, it deadlocks.

2.2.1 IRQ-Safe Spin Locks

The Linux kernel features several spin lock variants that are safe for using with interrupts. A critical section in process context is guarded by `spin_lock_irq` and `spin_unlock_irq`, while critical sections in interrupt handlers are guarded by the normal `spin_lock` and `spin_unlock`. The only difference between these functions is, that the irq-safe versions of `spin_lock` disable all interrupts on the local CPU for the critical section. The possibility of a deadlock is therefore eliminated.

Figure 1 shows how two CPUs interact when trying to acquire the same irq-safe spin lock.

While CPU 1 (in process context) is holding the lock, any incoming interrupt requests on CPU 1 are stalled until the lock is released. An interrupt on CPU 2 busy waits on the lock, but it does not deadlock. After CPU 1 releases the lock, CPU 2 (in interrupt context) obtains it, and CPU 1 (now executing the interrupt handler) waits for CPU 2 to release it.

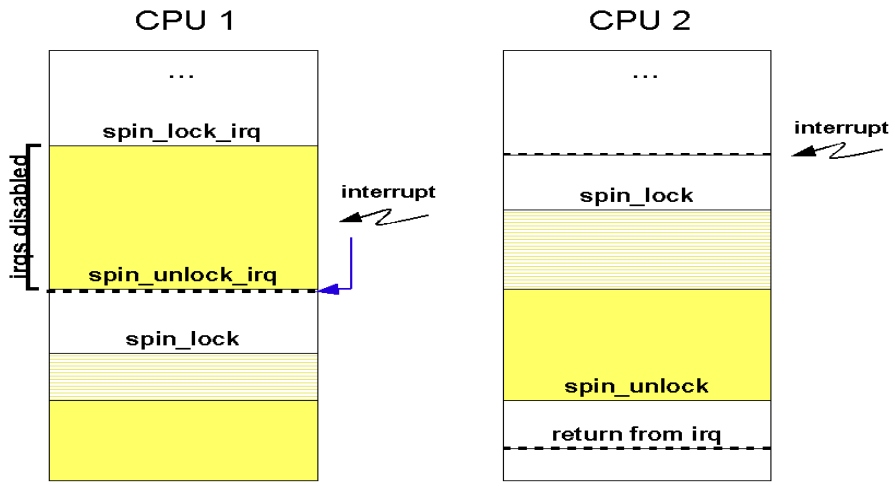

Figure 1: IRQ-safe spin locks

2.2.2 Enhancements of the simple Spin Lock

Sometimes it is wanted to allow spin locks to be nested. To do so, the spin lock is extended by a nesting counter and a variable indicating which CPU holds the lock. If the lock is held, the lock function checks if the current CPU is the one holding the lock. In this case, the nesting counter is incremented and it exits from the spin loop. The unlock function decrements the nesting counter. The lock is released when the unlock function has been called the same number of times the lock function was called before. This kind of spin lock is called a *recursive lock*.

Locks can also be modified to allow blocking. Linux' and FreeBSD's big kernel lock is dropped if the process holding it sleeps (blocks), and reacquired when it wakes up.

Solaris 2.x provides a type of locking known as *adaptive locks*. When one thread attempts to acquire one of these that is held by another thread, it checks to see if the second thread is active on a processor. If it is, the first thread spins. If the second thread is blocked, the first thread blocks as well.

2.3 Semaphores (mutex)

Aside from the classical use of semaphores explained in section 1.1, semaphores (initialized with a counter value of 1) can also be used for protecting critical sections. For performance reasons, semaphores used for this kind of work are often realized as a separate primitive, called mutex, that replaces the counter with a simple lock status flag.

Using mutexes instead of spin locks is productive, if the critical section takes longer than a context switch. Else, the overhead of blocking compared to busy waiting for a lock to be released is worse. On the pro side, mutexes imply a kind of fairness, while processes could

starve on heavily contended spin locks. Mutexes can not be used in interrupts, because it is generally not allowed to block in interrupt context.

A semaphore is a complex shared data structure itself, and must therefore be protected by an own spin lock.

2.4 Reader/Writer Locks

As reading a data structure does not affect the integrity of data, it is not necessary to mutually exclude two processes from reading the same data at the same time. If a data structure is read often, allowing readers to operate in parallel is a great advantage for SMP software.

An rwlock keeps count of the readers currently holding a read-only lock and has a queue for both waiting writers and waiting readers. If the writer queue is empty, new readers may grab the lock. If a writer enters the scene, it has to wait for all readers to complete, then it gets an exclusive lock. Meanwhile arriving writers or readers are queued until the write lock is dropped. Then, all readers waiting in the queue are allowed to enter, and the game starts anew (waiting writers are put on hold after a writer completes to prevent starvation of readers). Figure 2 shows a typical sequence.

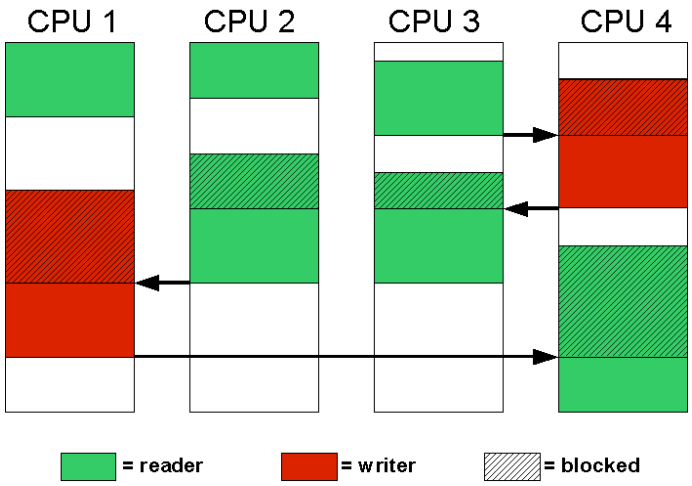

Figure 2: reader/writer lock

The rwlock involves a marginal overhead, but *should* yield almost linear scalability for read-mostly data structures (we will see about this later).

3 Locking Granularity in SMP Kernels

3.1 Giant Locking

The designers of the Linux operating systems did not have to worry much about mutual exclusion in their uniprocessor kernels, because they made the whole kernel non-preemptive (see section 1.1). The first Linux SMP kernel (version 2.0) used the most simple approach to make the traditional UP kernel code work on multiple CPUs: It protected the

whole kernel with a single lock, the big kernel lock (BKL). The BKL is a spin lock that could be nested and is blocking-safe (see section 2.2.2).

There could be no two CPUs in the kernel at the same time. The only advantage of this was that the rest of the kernel could be left unchanged.

3.2 Coarse-grained Locking

In Linux 2.2, the BKL was removed from the kernel entry points, and each subsystem was protected by an own lock. Now, a file system call would not have to wait for a sound driver routine or a network subsystem call to finish. Still, it was not data that was protected by the locks, bot rather concurrent function calls that were sequentialized. Also, the BKL could not be removed from all modules, because it was often unclear which data it protected. And data protected by the BKL could be accessed *anywhere* in the kernel.

3.3 Fine-grained Locking

Fine-grained locking means: Individual data structures, not whole subsystems or modules, are protected by their own locks. The degree of granularity can be increased from locks protecting big data structures (like, for example, a whole file system or the whole process table) to locks protecting individual data structures (for example, a single file or a process control block) or even single elements of a data structure. Fine-grained locking was introduced in the Linux 2.4 kernel series, and has been furthered in the 2.6 series.

Fine-grained locking has also been introduced into the FreeBSD operating system by the SMPng team, into the Solaris kernel, and into the Windows NT kernels as well.

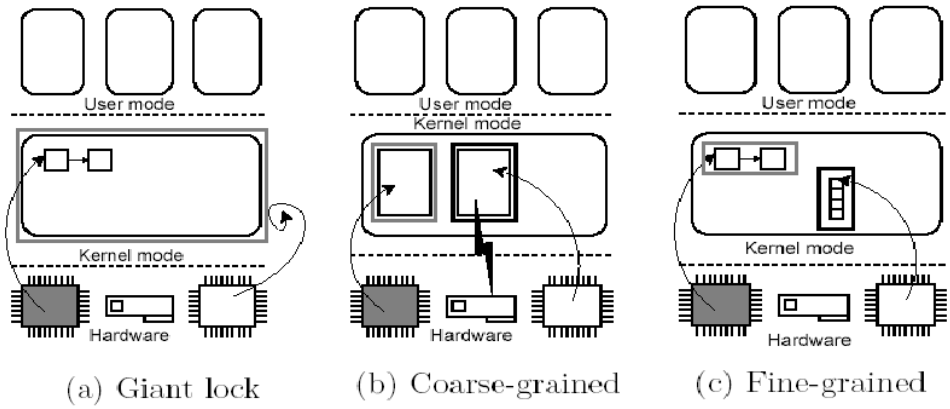

Figure 3: Visual description of locking granularity in OS kernels [Kag05]

Unfortunately, the BKL is still not dead. Changes to locking code had to be implemented very cautiously, as to not bring in hard to track down deadlock failures. So, every time a BKL was considered useless for a piece of code, it was moved into the functions this code called, because it was not always obvious if these functions relied on the BKL. Thus, the occurrences of BKL increased even more, and module maintainers did not always react to calls to remove the BKL from their code.

4 Performance Considerations

The Linux 2.0.40 contains a total of 17 BKL calls, while the Linux 2.4.30 kernel contains a total of 226 BKL, 329 spin lock and 121 rwlock calls. The Linux 2.6.11.7 kernel contains 101 BKL, 1717 spin lock and 349 rwlock calls, as well as 56 seq lock and 14 RCU (more on these synchronisation mechanisms later) (numbers taken from [Kag05]).

The reason, why the Linux programmers took so much work upon them, is that coarse-grained kernels scale poorly on more than 3-4 CPUs. The optimal performance for a n-CPU SMP system is n times the performance of a 1-CPU system of the same type. But this optimal performance can only be achieved if all CPUs are doing productive work all the time. Busy waiting on a lock wastes time, and the more contended a lock is, the more processors will likely busy wait to get it.

Hence, the kernel developers run special lock contention benchmarks to detect which locks have to be split up to distribute the invocations on them. The lock variables are extended by a lock-information structure that contains a counter for *hits* (successful attempts to grab a lock), misses (unsuccessful attempts to grab a lock), and *spins* (total of waiting loops) [Cam93]. The number of spins/misses shows how contended a lock is. If this number is high, processes waste a lot of time waiting for the lock.

		hits	misses	spins	spins/miss
_PageTableLockInfo	1	9,656	80	9,571	120
_DispatcherQueueLockInfo	1	49,979	382	30,508	80
_SleepHashQueueLockInfo	1	25,549	708	56,192	79

Figure 4: Extract from a lock-contention benchmark on Unix SVR4/MP [Cam93]

Measuring lock contention is a common practice to look for bottlenecks. Bryant and Hawkes wrote a specialized tool to measure lock contention in the kernel, which they used to analyze filesystem performance [Bry02]. Others [Kra01] focused on contention in the 2.4.x scheduler, which has since been completely rewritten. Today, the Linux scheduler mostly operates on per-CPU ready queues and scales fine up to 512 CPUs.

Locking is most pronounced with applications that access shared resources, such as the virtual filesystem (VFS) and network, and applications that spawn many processes. Etison et. al. used several benchmarks that stress these subsystems as an example of how the KLogger kernel logging and analysis tool can be used for measuring lock contention , using varying degrees of parallelization: They measured the percentage of time spent spinning on locks during make running a parallel compilation of the Linux kernel, Netperf (a network performance evaluation tool), and an Apache web server with Perl CGI being stress-tested [Eti05] (see Figure 5).

Measurements like these help to spot and eliminate locking bottlenecks.

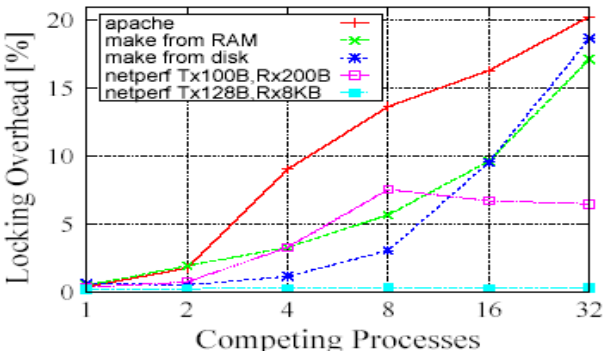

Figure 5: Percentage of cycles spent on spinning on locks for each of the test applications [Eti05]

4.1 Scalability

Simon Kågström made similar benchmarks to compare the scalability of the Linux kernel on 1-8 CPUs from version 2.0 to 2.6. He measured the relative speedup in regard to the (giant locked) 2.0.40 UP kernel with the Postmark benchmark (Figure 6).

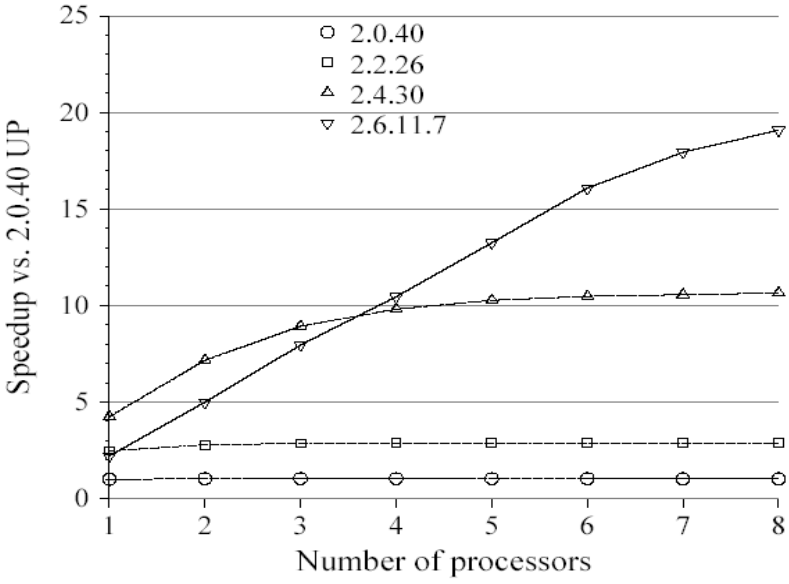

Figure 6: Postmark benchmark of several Linux kernels, CONFIG_SMP=y [Kag05]

The result of this benchmark is not surprising. As we can see, the more we increase locking granularity (Linux 2.6), the better the system scales. But how far can we increase locking granularity?

Of course, we cannot ignore the increasing complexity induced by finer locking granularity. As more locks have to be held to perform a specific operation, the risk of deadlock increases. There is an ongoing discussion in the Linux community about how much locking hierarchy is too much. With more locking comes more need for documentation of locking order, or need for tools like deadlock analyzers. Deadlock faults are among the most difficult to come by.

The overhead of locking operations matters as well. The CPU does not only spend time in a critical section, it also takes some time to acquire and to release a lock. Compare the graph of the 2.6 kernel with the 2.4 kernel for less than four CPUs: The kernel with more locks is the slower one. The efficiency of executing a critical section can be mathematically expressed as: Time within critical section / (Time within critical section + Time to acquire lock + Time to release lock). If you split a critical section into two, the time to acquire and release a lock can be roughly multiplied by two.

Surprisingly, even one time the acquisition of a lock is generally one time too much, and the performance penalty of a simple lock acquisition, even if successful at the first attempt, is becoming worse and worse. To understand why, we have to forget the basic model of a simple scalar processor without caches and look at today's reality.

4.2 Performance Penalty of Lock Operations

Image 1 shows typical instruction costs of several operations on a 8-CPU 1.45 GHz PPC system[1]. The gap between normal instructions, cache-hitting memory accesses (not listed here; they are generally 3-4 times faster than an atomic increment operation) and a lock operation becomes obvious.

Operation	Nanoseconds
Instruction	0.24
Clock Cycle	0.69
Atomic Increment	42.09
Cmpxchg Blind Cache Transfer	56.80
Cmpxchg Cache Transfer and Invalidate	59.10
SMP Memory Barrier (eieio)	75.53
Full Memory Barrier (sync)	92.16
CPU-Local Lock	243.10

Image 1: Instruction costs on a 8-CPU 1.45GHz PPC system [McK05]

Let us look at the architecture of today's SMP systems and it's impact on our spin lock.

4.2.1 Caches

As CPU power has increased roughly by factor 2 each year, memory speeds have not kept pace, and increased by only 10 to 15% each year. Thus, memory operations impose a big performance penalty on todays' computers.

1 If you wonder why an instruction takes less time then a CPU cycle, remember that we are looking at a 8-CPU SMP system, and view these numbers as "typical instruction cost".

As a consequence of this development, small SRAM caches were introduced, which are much faster than main memory. Due to temporal and spatial locality of reference in programs (see [Sch94] for explanation), even a comparatively small cache achieves hit ratios of 90% and higher. On SMP systems, each processor has its own cache. This has the big advantage that cache hits cause no load on the common memory bus, but it introduces the problem of cache consistency.

When a memory word is accessed by a CPU, it is first looked up in the CPU's local cache. If it is not found there, the whole cache line[2] containing the memory word is copied into the cache. This is called a cache miss (to increase the number of cache hits, it is thus very advisable to align data along cache lines in physical memory and operate on data structures that fit within a single cache line). Subsequent read accesses to that memory address will cause a cache hit. But what happens on a write access to a memory word that lies in the cache? This depends on the "write policy".

"Write through" means that after every write access, the cache line is written back to main memory. This insures consistency between the cache and memory (and between all caches of a SMP system, if the other caches snoop the bus for write accesses), but it is also the slowest method, because a memory access is needed on every write access to a cache line.

The "write back" policy is much more common. On a write access, data is not written back to memory immediately, but the cache line gets a "modified" tag. If a cache line with a modified tag is finally replaced by another line, it's content is written back to memory. Subsequent write operations hit in the cache as long as the line is not removed.

On SMP systems, the same piece from physical memory could lie in more then one cache. The SMP architecture needs a protocol to insure consistency between all caches. If two CPUs want to read the same memory word from their cache, everything goes well. In addition, both read operations can execute at the same time. But if two CPUs wanted to write to the same memory word in their cache at the same time, there would be a modified version of this cache line in both caches afterwards and thus, two versions of the same cache line would exist. To prevent this, a CPU trying to modify a cache line has to get the "exclusive" right on it. With that, this cache line is marked invalid in all other caches. Another CPU trying to modify a cache line has to wait until CPU 1 drops the exclusive right, and has to re-read the cache line from CPU 1's cache.

Let us look at the effects of the simple spin lock code from Listing 3, if a lock is held by CPU 1, and CPU 2 and 3 wait for it:

The lock variable is set by the test_and_set operation on every spinning cycle. While CPU 1 is in the critical section, CPU 2 and 3 constantly read and write to the cache line containing the lock variable. The line is constantly transferred from one cache to the other, because both CPUs must acquire an exclusive copy of the line when they test-and-set the lock variable again. This is called "cache line bouncing", and it imposes a big load on the memory bus. The impact on performance would be even worse if the data protected by the lock was also lying in the same cache line.

We can however modify the implementation of the spin lock to fit the functionality of a

2 To find data in the cache, each line of the cache (think of the cache as a spreadsheet) has a tag containing it's address. If one cache line would consist of only one memory word, a lot of lines and thus, a lot of address tags, would be needed. To reduce this overhead, cache lines contain usually about 32-128 bytes, accessed by the same tag, and the least significant bits of the address serve as the byte offset within the cache line.

cache.

The atomic read-modify-write operation cannot possibly acquire the lock while it is held by another processor. It is therefore unnecessary to use such an operation until the lock is freed. Instead, other processors trying to acquire a lock that is in use can simply read the current state of the lock and only use the atomic operation once the lock has been freed. Listing 4 gives an alternate implementation of the lock function using this technique.

```
void lock(volatile lock_t *lock_status)
{
      while (test_and_set(lock_status) == 1)
            while (*lock_status == 1); // spin
}
```

Listing 4: Spin lock implementation avoiding excessive cache line bouncing [Sch94]

Here, one attempt is made to acquire the lock before entering the inner loop, which then waits until the lock is freed. If the lock is already taken again on the test_and_set operation, the CPU spins again in the inner loop. CPUs spinning in the inner loop only work on a shared cache line and do not request the cache line exclusive. They work cache-local and do not waste bus bandwidth. When CPU 1 releases the lock, it marks the cache line exclusive and sets the lock variable to zero. The other CPUs re-read the cache line and try to acquire the lock again.

Nevertheless, spin lock operations are still very time-consumptive, because they usually involve at least one cache line transfer between caches or from memory.

4.2.2 Memory Barriers

With the superscalar architecture, parallelism was introduced into the CPU cores. In a superscalar CPU, there are several functional units of the same type, along with additional circuitry to *dispatch* instructions to the units. For instance, most superscalar designs include more than one arithmetic-logical unit. The dispatcher reads instructions from memory and decides which ones can be run in parallel, dispatching them to the two units. The performance of the dispatcher is key to the overall performance of a superscalar design: The units' pipelines should be as full as possible. A superscalar CPU's dispatcher hardware therefore reorders instructions for optimal throughput. This holds true for load/store operations as well.

For example, imagine a program that adds two integers from main memory. The first argument that is fetched is not in the cache and must be fetched from main memory. Meanwhile, the second argument is fetched from the cache. The second load operation is likely to complete earlier. Meanwhile, a third load operation can be issued. The dispatcher uses interlocks to prohibit that the add operation is issued before the load operations it depends on are finished.

Also, most modern CPUs sport a small register set called store buffers, where several store operations are gathered to be executed at once at a later time. They can be buffered in-order (which is called total store ordering) or – as common with superscalar CPUs – out of order (partial store ordering). In short: As long as a load or store operation does not access the

same memory word as a prior store operation (or vice versa), they can be executed in any possible order by the CPU.

This needs further measures to ensure correctness of SMP code. The simple atomic list insert code from section 2.1 could be executed as shown in Figure 7.

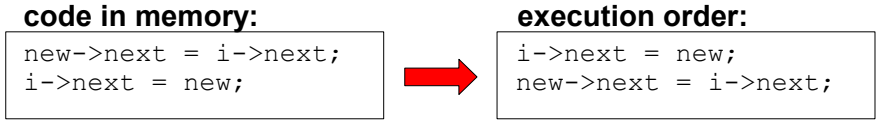

Figure 7: Impact of non sequential memory model on atomic list insertion algorithm

The method requires the new node's next pointer (and all it's data) to be initialized *before* the new element is inserted at the list's head. If these instructions are out of order, the list will be in an inconsistent state until the second instruction completes. Meanwhile, another CPU could traverse the list, the thread could be preempted, etc.

It is not necessary that both operations are executed right after each other, but it is important that the first one was executed before the second. To force finishing of all read or write operations in the instruction pipeline before the next operation is fetched, superscalar CPUs have so-called memory barrier instructions. We distinguish read memory barriers (wait until all pending read operations have completed), write memory barriers, and memory barriers (wait until all pending memory operations have completed, read and write). Correct code would read:

```
new->next = i->next;
smp_wmb(); // write memory barrier!
i->next = new;
```

Listing 5: Correct code of atomic list insertion on machines without sequential memory model

Instruction reordering can also cause operations in a critical section to "bleed out" (Figure 8). The line that claims to be in a critical section is obviously not, because the operation releasing the lock variable was executed earlier. Another CPU could long ago have altered a part of the data, with the results being unpredictable.

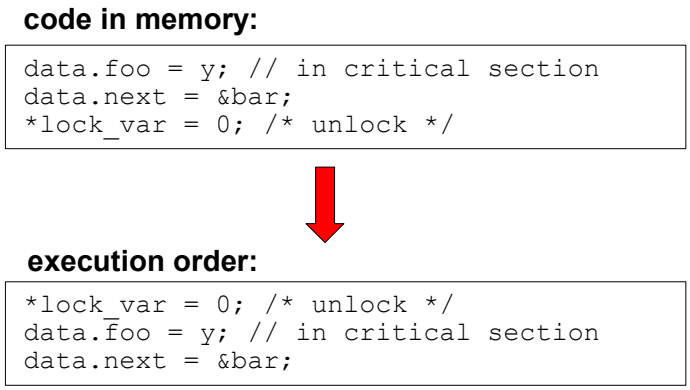

Figure 8: Impact of weak store ordering on critical sections

To prevent this, we have to alter our locking operations again (note that it is not necessary to prevent load/store operations prior to the critical section to "bleed" into it. And of course, dispatcher units do not override the logical instruction flow, so every operation in the critical section will be executed after the CPU exits that while loop):

```
void lock(volatile lock_t *lock_var)
{
      while (test_and_set(lock_var) == 1)
            while (*lock_var == 1); // spin
}

void unlock(volatile lock_t *lock_var)
{
      mb(); // read-write memory barrier
      *lock_var = 0;
}
```

Listing 6: Spin lock with memory barrier

A memory barrier flushes the store buffer and stalls the pipelines (to carry out all pending read/write operations before new ones are executed), so it negatively impacts the performance proportional to the number of pipeline stages and number of functional units. This is why the memory barrier operations take so much time in the chart presented earlier. Atomic operations take so long because they also flush the store buffer in order to be carried out immediately.

4.2.3 Hash-Table Benchmark

Below are the results of a benchmark that performed search operations on a hash table with a dense array of buckets, doubly linked hash chains, and one element per hash chain. Under the locking designs tested for this hash table were: Global spin lock, global reader/writer

lock, per-bucket spin lock and rwlock and Linux' big reader lock. Especially the results for the allegedly parallel reader/writer locks seem surprising, but only support the things said in the last two sections: The locking instructions' overhead thwarts any parallelism.

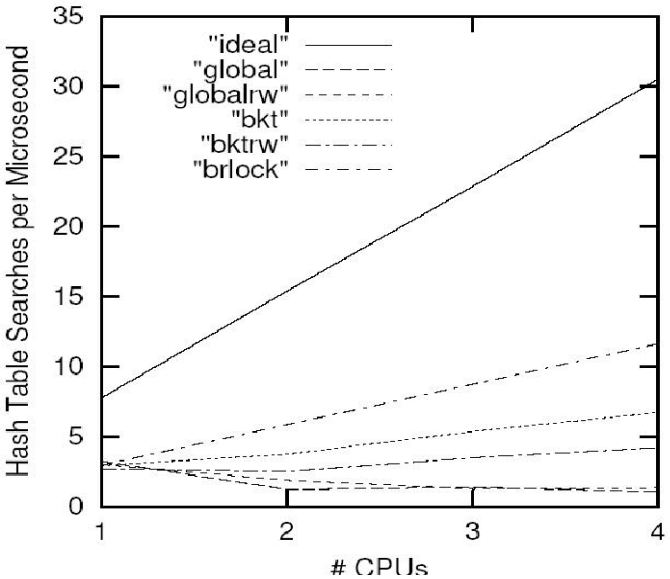

Image 2: Performance of several locking strategies for a hash table ([McK01])

The explanation for this is now rather simple (Figure 9): The acquisition and release of rwlocks take so much time (remember the cache line bouncing etc.), that the actual critical section is not executed parallel any more.

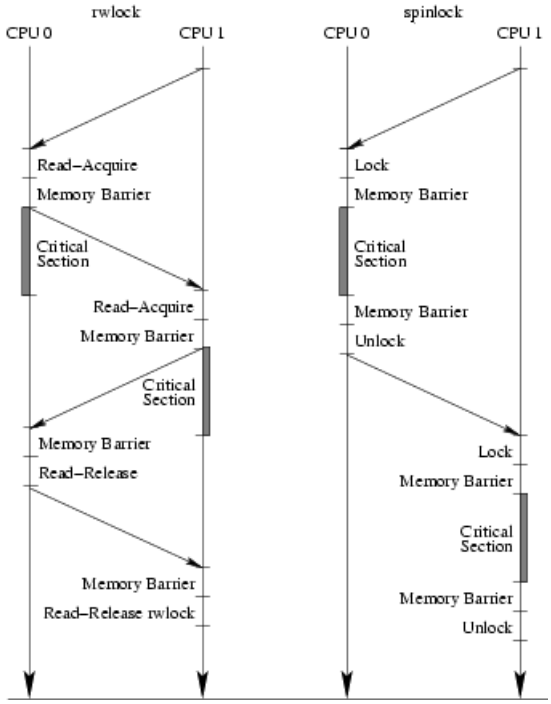

Figure 9: Effects of cache line bouncing and memory synchronisation delay on rwlock's efficiency ([McK03])

5 Lock-Avoiding Synchronization Primitives

Much effort has been put in developing synchronisation primitives that avoid locks. Lock-free and wait-free synchronisation also plays a major part in real time operating systems, where time guarantees must be given. Another way to reduce lock contention is by using per-CPU data.

Two new synchronisation mechanisms that get by totally without locking or atomic operations on the reader side were introduced into the Linux 2.6 kernel to address the above issues: The SeqLock and the Read-Copy-Update mechanism. We will discuss them in particular.

5.1 Seq Locks

Seq locks (short for sequence locks) are a variant of the reader-writer lock, based on spin locks. They are intended for short critical sections and tuned for fast data access and low latency.

In contrast to the rwlock, a sequence lock warrants writer access immediately, regardless of any readers. Writers have to acquire a writer spin lock which provides mutual exclusion for multiple writers. They then can alter the data without paying regard to possible readers.

Therefore, readers also do not need to acquire any lock to synchronize with possible

writers. Thus, a read access generally does not acquire a lock, but readers are in charge to check if they read valid data: If a write access took place while the data was read, the data is invalid and has to be read again. The identification of write accesses is realized with a counter (see Figure 10). Every writer increments this zero-initialized counter once before he changes any data, and again after all changes are done. The reader reads the counter value before he reads the data, then compares it to the current counter value after reading the data. If the counter value has increased, the reading was tampered by one or more concurrent writers and the data has to be read again. Also, if the counter value was uneven at the beginning of the read-side critical section, a writer was in progress while the data was read and it has to be discarded. So, strictly speaking, the while loop's condition is `((count_pre != count_post) && (count_pre % 2 == 0))`.

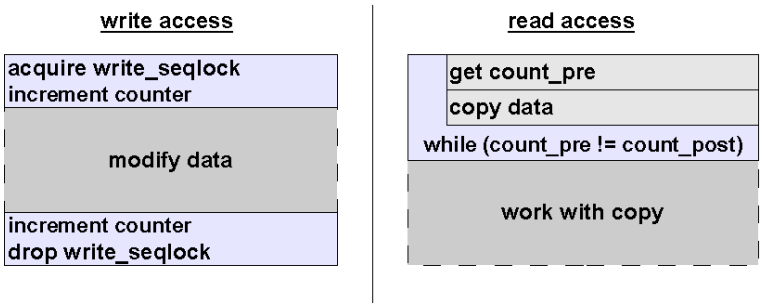

Figure 10: Seq Lock schematics (figure based upon [Qua04])

In the worst case, the readers would have to loop infinitely if there was a non-ending chain of writers. But under normal conditions, the readers read the data successfully within a few, if not only one tries. By minimizing the time spent in the read-side critical section, the probability of being interrupted by a writer can be reduced greatly. Therefore, it is part of the method to only copy the shared data in the critical section and work on it later.

```
unsigned long seq;

do {
   seq = read_seqbegin(&time_lock);
   now = time;
} while( read_seqretry(&time_lock,seq) );
   // value in "now" can now be used
```

Listing 7: Seq Lock: Read-side critical section

Listing 7 shows how to read shared data protected by the seq lock functions of the Linux kernel. `time_lock` is the seq lock variable, but `read_seqbegin` and `read_seqretry` only read the seq lock's counter and do not access the lock variable.

5.2 The Read-Copy-Update Mechanism

As you can see, synchronisation is a mechanism and a coding convention. The coding convention for mutual exclusion with a spin lock is, that you have to hold a lock before you access the data protected by it, and that you have to release it after you are done. The

coding convention for non-blocking synchronisation is, that every data manipulation only needs a single atomic operation (e.g. CASW, CASW2 or our atomic list update example).

The RCU mechanism is based on something called quiescent states. A quiescent state is a point in time, where a process that has been reading shared data does not hold any references to this data any more. With the RCU mechanism, processes can enter a read-side critical section any time and can assume that the data they read is consistent as long as they work on it. But after a process leaves it's read-side critical section, it *must* not hold any references to the data any longer. The process enters the quiescent state.

This imposes some constraints on how to update shared data structures. As readers do not check if the data they read is consistent (as in the SeqLock mechanism), writers have to apply *all* their changes with *one atomical operation*. If a reader read the data before the update, it sees the old state, if the reader reads the data after the update, it sees the new state. Of course, readers should read data *once* and than work with it, and not read the same data several times and than fail if they read differing versions. Consider a linked list protected by RCU. To update an element of the list (see Figure 11), the writer has to read the old element's contents, make a copy of them ***(1)***, update this element ***(2)***, and then exchange the old element with the new one with an atomic operation (writing the new element's address to the previous element's next pointer) ***(3)***.

As you can see, readers *could* still read stale data even after the writer has finished updating, if they entered the read-side critical section before the writer finished ***(4)***. Therefore, the writer cannot immediately delete the old element. It has to defer the destruction, until all processes that were in a read-side critical section at the time the writer finished have dropped their references to the stale data ***(5)***. Or in other words, entered the quiescent state. This time span is called the *grace period*. After that time, there can be readers holding references to the data, but none of them could possibly reference the old data, because they started at a time when the old data was not visible any more. The old element can be deleted ***(6)***.

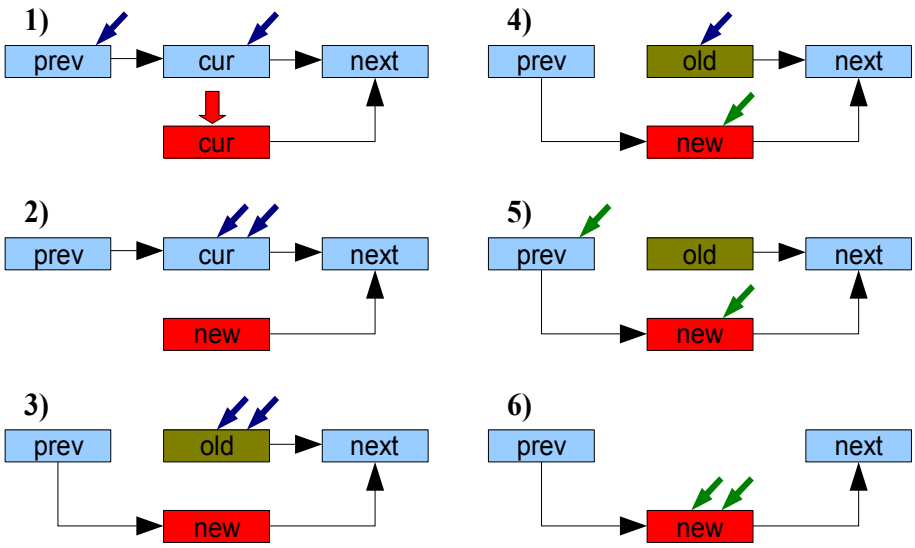

Figure 11: The six steps of the RCU mechanism

The RCU mechanism requires data that is stored within some sort of container that is referenced by a pointer. The update step consists of changing that pointer. Thus, linked lists are the most common type of data protected by RCU. Insertion and deletion of elements is done like presented in section 2.1. Of course, we need memory barrier operations on machines with weak ordering. More complex updates, like sorting a list, need some other kind of synchronisation mechanism. If we assume that readers traverse a list in search of an element once, and not several times back and forth (as we assumed anyways), we can also use doubly linked lists (see Listing 8).

```
static inline void __list_add_rcu(struct list_head * new,
      struct list_head * prev, struct list_head * next)
{
      new->next = next;
      new->prev = prev;
      smp_wmb();
      next->prev = new;
      prev->next = new;
}
```

Listing 8: extract from Linux 2.6.14 kernel, list.h

The RCU mechanism is optimal for read-mostly data structures, where readers can tolerate stale data (it is for example used in the Linux routing table implementation). While readers generally do not have to worry about much, things get more complex on the writer's side. First of all, writers have to acquire a lock, just like with the seqlock mechanism. If they would not, two writers could obtain a copy of a data element, perform their changes, and then replace it. The data structure will still be intact, but one update would be lost. Second, writers have to defer the destruction of an old version of the data until sometime. When exactly is it safe to delete old data?

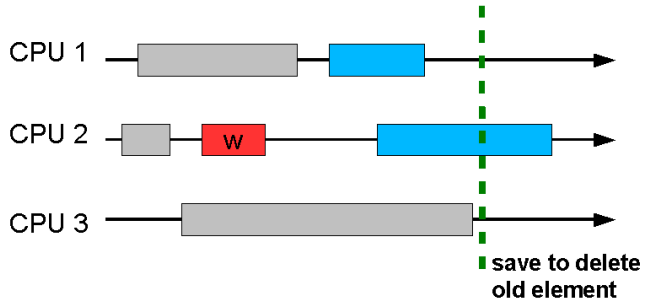

Figure 12: RCU, Grace Period: After all processes that entered a read-side critical section (gray) before the writer (red) finished have entered a quiescent state, it is save delete an old element

After all readers that were in a read-side critical section at the time of the update have left their critical section, or, entered a quiescent state (Figure 12). A simple approach would be to take a counter that indicates how many processes are within a read-side critical section, and defer destruction of all stale versions of data elements until that time. But as you can see, later readers are not taken into account, so this approach fails. We could also include a reference counter in every data element, if the architecture features an atomic increment operation. Every reader would increment this reference counter as it gets a reference to the

data element, and decrement it when the read-side critical section completes. If an old data element has a refcount of zero, it can be deleted [McK01]. While this solves the problem, it reintroduces the performance issues of atomic operations that we wanted to avoid.

Let us assume that a process in a RCU read-side critical section does not yield the CPU. This means: Preemption is disabled while in the critical section and no functions that might block must be used [McK04]. Then no references can be held across a context switch, and we can fairly assume a CPU that has gone through a context switch to be in a quiescent state. The earliest time when we can be absolutely sure that no process on any other CPU is still holding a reference to stale data, is after every other CPU has gone through a context switch at least once after the writer finished. The writer has to defer destruction of stale data until then, either by waiting or by registering a callback function that frees the space occupied by the stale data. This callback function is called after the grace period is over. The Linux kernel features both variants.

A simple mechanism to detect when all CPUs have gone through a context switch is to start a high priority thread on CPU 1 that repeatedly reschedules itself on the next CPU until it reaches the last CPU. This thread then executes the callback functions or wakes up any processes waiting for the grace period to end. There are more effective algorithms out there to detect the end of a grace period, but these are out of the scope of this document.

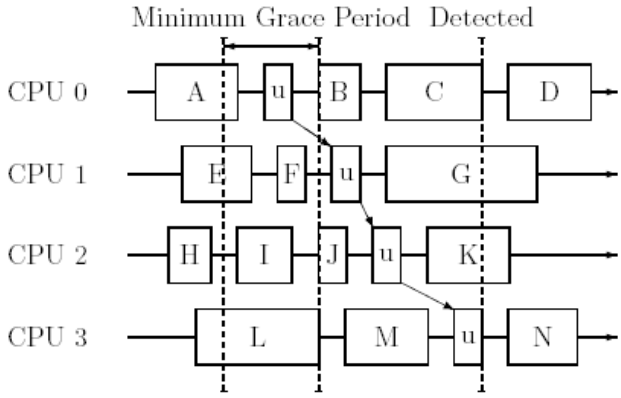

Figure 13: Simple detection of a grace period: Thread "u" runs once on every CPU [McK01]

Listing 9 presents the Linux RCU API (without the Linux RCU list API). Note that, while it is necessary to guard read-side critical sections with `rcu_read_lock` and `rcu_read_unlock`, the only thing these functions do (except for visually highlighting a critical section) is disabling preemption for the critical section. If the Linux kernel is compiled with `PREEMPT_ENABLE=no`, they do nothing. Write-side critical sections are protected by spin_lock() and spin_unlock(), and wait for the grace period afterwards with `synchronize_kernel()` or register a callback function to destroy the old element with `call_rcu()`.

```
void synchronize_kernel(void);
void call_rcu(struct rcu_head *head,
              void (*func)(void *arg),
              void *arg);
struct rcu_head {
        struct list_head list;
        void (*func)(void *obj);
        void *arg;
};
void rcu_read_lock(void);
void rcu_read_unlock(void);
```

Listing 9: The Linux 2.6 RCU API functions

The RCU mechanism is widely believed to have been developed at Sequent Computer Systems, who were then bought by IBM, who holds several patents on this technique. The patent holders have given permission to use this mechanism under the GPL. Therefore, Linux is currently the only major OS using it. RCU is also part of the SCO claims in the SCO vs. IBM lawsuit.

6 Conclusion

The introduction of SMP systems has greatly increased the complexity of locking in OS kernels. In order to strive for optimal performance on all platforms, the operating system designers have to meet conflictive goals: Increase granularity to increase scalability of their kernels, and reduce using of locks to increase the efficiency of critical sections, and thus the performance of their code. While a spin lock *can* always be used, it is not always the right tool for the job. Non-blocking synchronisation, Seq Locks or the RCU mechanism offer better performance than spin locks or rwlocks. But these synchronisation methods require more effort than a simple replacement. RCU requires a complete rethinking and rewriting of the data structures it is used for, and the code it is used in.

It took half a decade for Linux from it's first giant locked SMP kernel implementation to a reasonably fine granular. This time could have been greatly reduced, if the Linux kernel had been written as a preemptive kernel with fine granular locking from the beginning. When it comes to mutual exclusion, it is always a good thing to think the whole thing through from the beginning. Starting with an approach that is ugly but works, and tuning it to a well running solution later, often leaves you coding the same thing twice, and experiencing greater problems than if you tried to do it nicely from the start.

As multiprocessor systems and modern architectures like superscalar, super pipelined, and hyperthreaded CPUs as well as multi-level caches become normalcy, simple code that looks fine at first glance can have severe impact on performance. Thus, programmers need to have a thorough understanding of the hardware they write code for.

Further Reading

This paper detailed on the performance aspects of locking in SMP kernels. If you are

interested in the implementation complexity of fine grained SMP kernels or experiences from performing a multiprocessor port, please refer to [Kag05]. For a more in-depth introduction to SMP architecture and caching, read Curt Schimmel's book ([Sch94]).

If you want to gain deeper knowledge of the RCU mechanism, you can start at Paul E. McKenney's RCU website (http://www.rdrop.com/users/paulmck/RCU).

References

[Bry02] R. Bryant, R. Forester, J. Hawkes: Filesystem performance and scalability in Linux 2.4.17. *Proceedings of the Usenix Annual Technical Conference*, 2002

[Cam93] Mark D. Campbell, Russ L. Holt: Lock-Granularity Analysis Tools in SVR4/MP. *IEEE Software*, March 1993

[Eti05] Yoav Etison, Dan Tsafrir et. al.: Fine Grained Kernel Logging with KLogger: Experience and Insights. *Hebrew University*, 2005

[Kag05] Simon Kågström: Performance and Implementation Complexity in Multiprocessor Operating System Kernels. *Blekinge Institute of Technology*, 2005

[Kra01] M. Kravetz, H. Franke: Enhancing the Linux scheduler. *Proceedings of the Ottawa Linux Symposium*, 2001

[McK01] Paul E. McKenney: Read-Copy Update. *Proceedings of the Ottawa Linux Symposium*, 2002

[McK03] Paul E. McKenney: Kernel Korner - Using RCU in the Linux 2.5 Kernel. *Linux Magazine*, 2003

[McK04] Paul E. McKenney: RCU vs. Locking Performance on Different Types of CPUs. *http://www.rdrop.com/users/paulmck/RCU/LCA2004.02.13a.pdf*, 2004

[McK05] Paul McKenney: Abstraction, Reality Checks, and RCU. *http://www.rdrop.com/users/paulmck/RCU/RCUintro.2005.07.26bt.pdf*, 2005

[Qua04] Jürgen Quade, Eva-Katharina Kunst: Linux Treiber entwickeln. *Dpunkt.verlag*, 2004

[Sch94] Curt Schimmel: UNIX Systems for Modern Architectures: Symmetric Multiprocessing and Caching for Kernel Programmers. *Addison Wesley*, 1994