

Effective Synchronization on Linux/NUMA Systems

by

Christoph Lameter

Revision: May 20th, 2005

© 2005 Silicon Graphics, Inc. All rights reserved.

This presentation and the corresponding paper may be found at
http://oss.sgi.com/projects/page_fault_performance/gelato

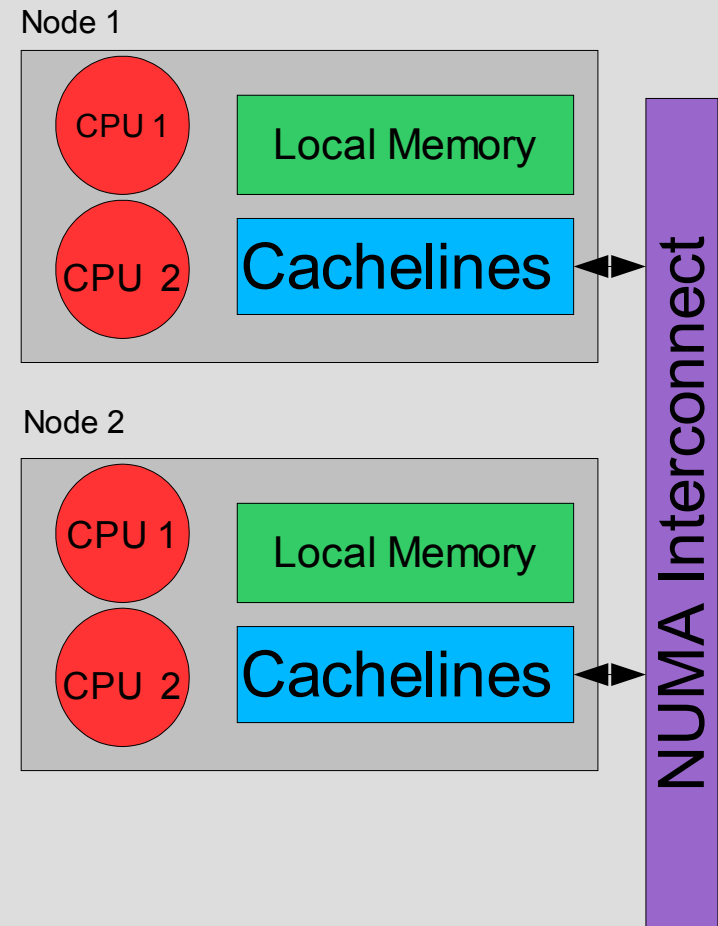
Effective locking is necessary for satisfactory performance on large Itanium based NUMA systems. Synchronization of parallel executing streams on NUMA machines is currently realized in the Linux kernel through a variety of mechanisms which include atomic operations, locking and ordering of memory accesses. Various synchronization methods may also be combined in order to increase performance.

Introduction

- Limits on processor clock rate
 - Future: *Multi-Core* and *NUMA* everywhere
 - Parallelism Itanium / Multi-Core
- *Synchronization Methods*
 - Critical Component for concurrency
 - Determines viable hardware scaling
- Outline
 - Existing synchronization on Linux / Itanium
 - Reasons for issues with lock contention arises
 - Hierarchical Backoff Locks on large NUMA systems.

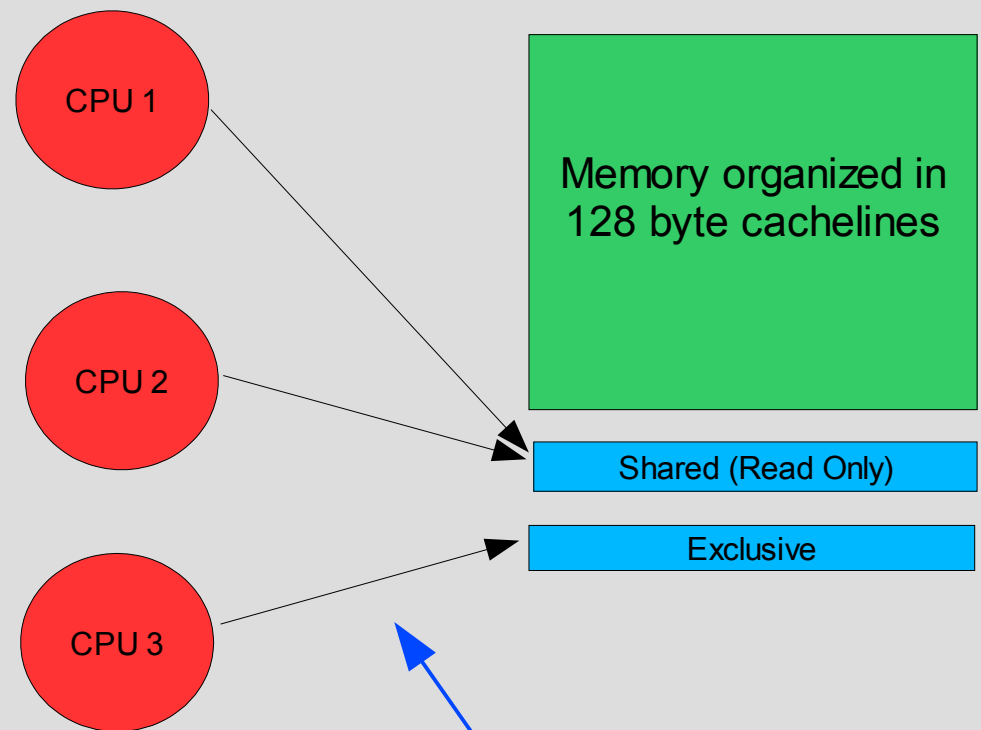
Basic Atomicity

- NUMA Multiprocessor Systems
 - NUMA interconnect
 - Hardware consistency protocol
- Node
 - Processor/ Memory
- Cache Line
- MESI type
 - Coherent view of memory
 - In Hardware



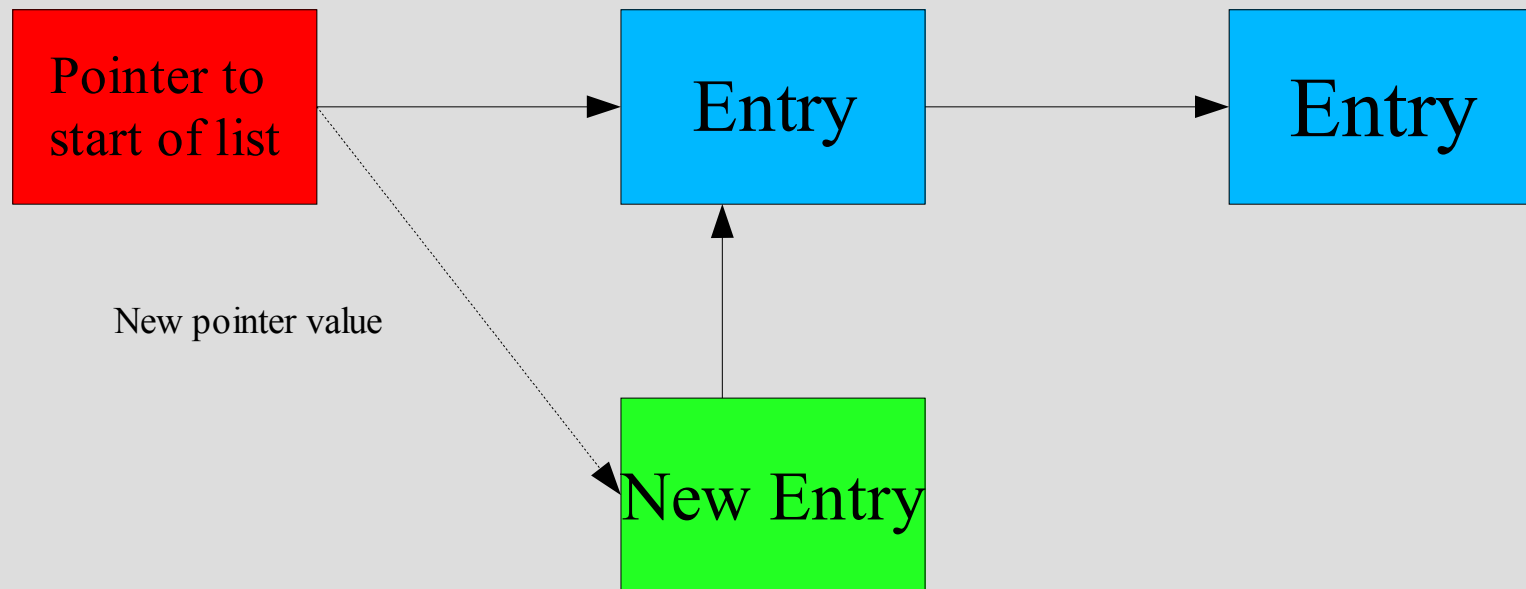
Cache Lines

- Modes of Cachelines
 - Shared
 - Exclusive
- Cache Lines
 - Efficiency
 - Optimization
 - Bouncing
- Special Operations
 - Read Modify Write



Atomic Operations can be performed on a cacheline if a cpu has exclusive ownership of a cache line

Atomic Loads and Stores



- 64 bit atomic operations
 - Alignment issues
- RCU functions in the Linux kernel
- A lockless insertion of a list element

Barriers and Acquire/Release

- Itanium Memory accesses
 - Unordered by nature
 - Necessity of ordering memory accesses
 - Memory Fence
 - Instructions with acquire / release semantics
 - Write and Read barriers
- Semaphore instructions
 - Necessity
 - Efficiency vs. atomic loads / stores

Linux RCU Lockless List Manipulation

- In include/linux/list.h
 - `list_add_rcu(struct list_head *new, *head)`
 - `list_del_rcu(struct list_head *entry)`
 - `list_for_each_entry_rcu(..)`
- Single writer/ multiple readers
 - Deferral of freeing objects
 - `rcu_read_lock`
 - `rcu_read_unlock`

```
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;
}

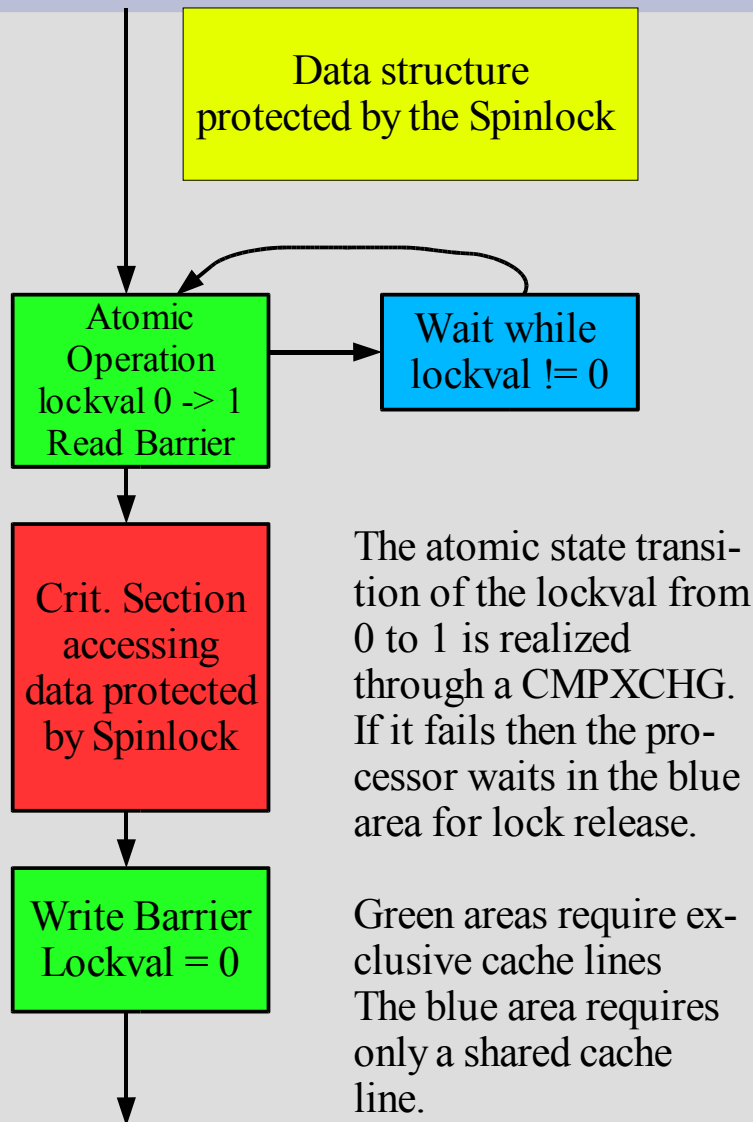
void list_add_rcu(struct list_head *new,
                 struct list_head *head)
{
    __list_add_rcu(new, head, head->next);
}
```

- Write exclusive requires a regular lock

Itanium Semaphore Instructions

- Read Modify Write cycles
 - exclusive cacheline
 - Non-speculative
 - Pipeline stalls
 - Acquire or release semantics
- Single processor effects a certain state change
 - Compare and Exchange CMPXCHG
 - Fetch and add FETCHADD
 - Exchange XCHG

The Spinlock Implementation



- Protected Data
- Critical Sections
- Locking
- Unlocking
- Exclusive Cache line use vs. Shared Cache line
- Bouncing Cachelines
- Spinlocks under contention

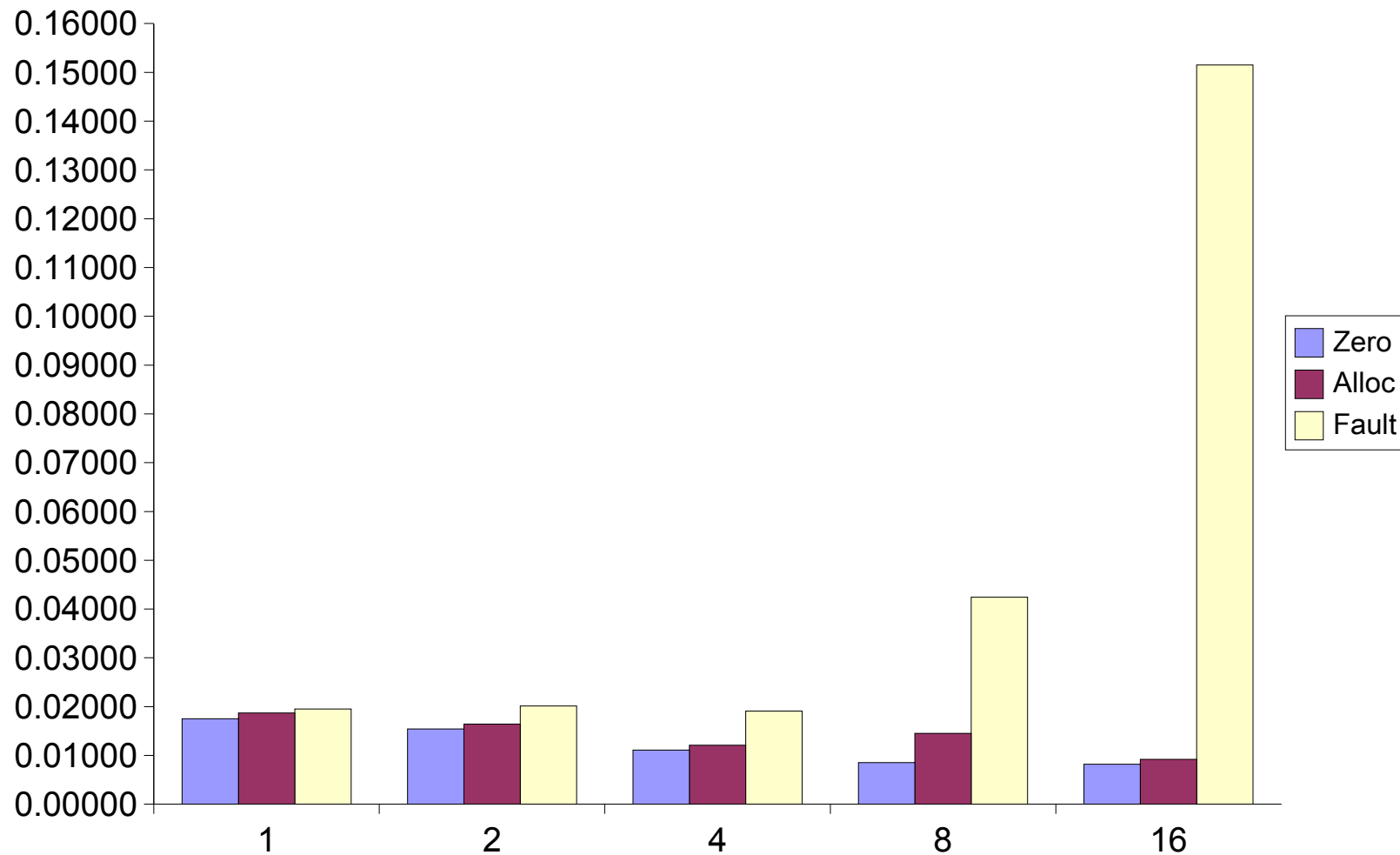
Spinlock Examples

- Spinlock Functions
- Sample Use

```
spin_lock(spinlock_t *lock);  
spin_unlock(spinlock_t *lock);
```

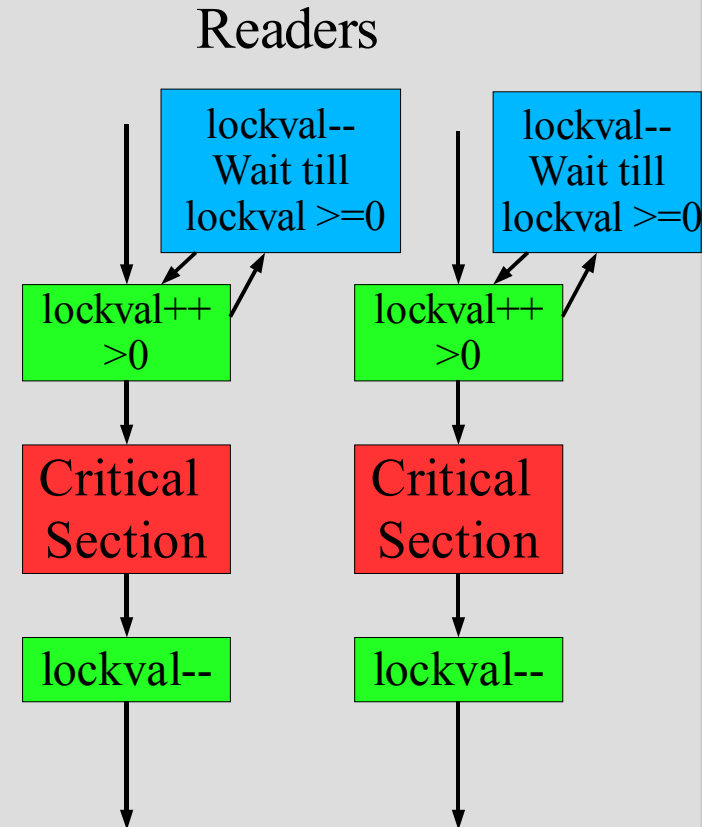
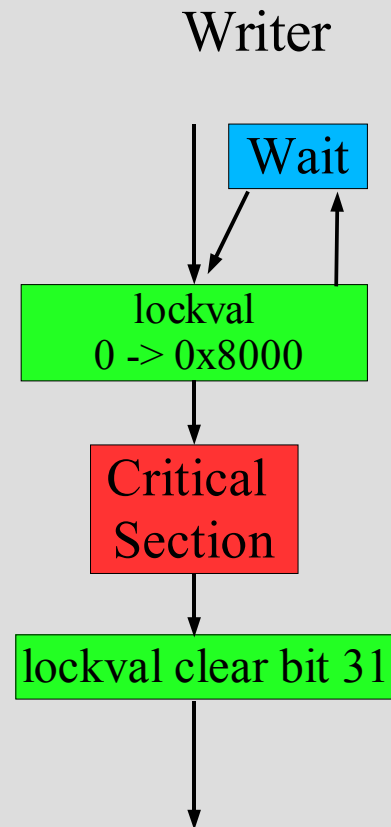
```
spin_lock(&mmlist_lock);  
list_add(&dst_mm->mmlist, &src_mm->mmlist);  
spin_unlock(&mmlist_lock);
```

Time in the Page Fault Handler



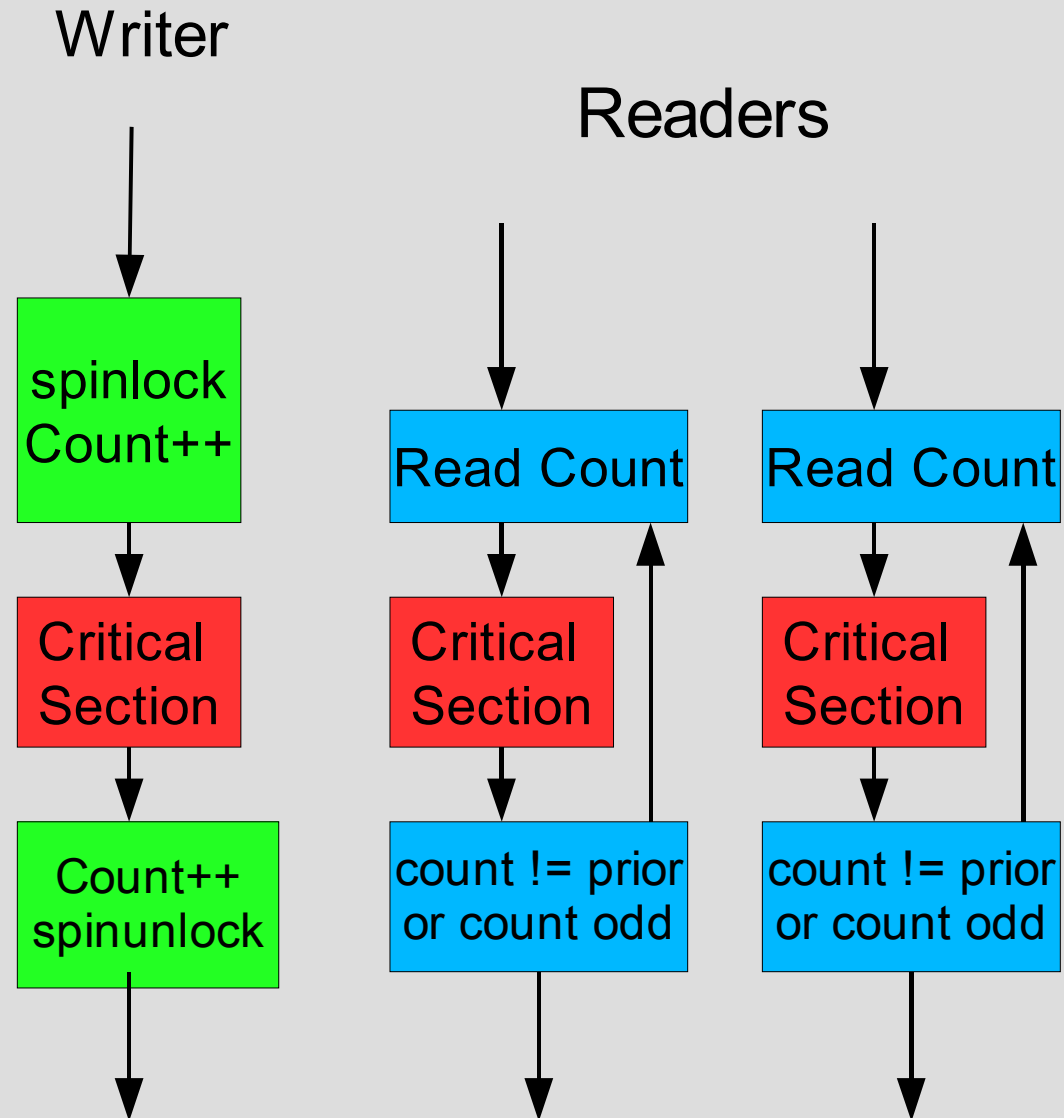
Reader/Writer Spinlocks

- Lock value
 - $>0 \rightarrow$ nr readers
 - <0 writer
 - 0 free
- Needs
 - 2x Cmpxchg
 - 2x Fetchadd
 - Clear Bit 31 (byte store instead?)
- Performance worse than regular spinlock



Sequence locks

- Most scalable lock
 - Is this a “lock”?
 - no write for readers
- Effort
 - Writer
 - 2xCmpxchg
 - 2xFetchadd
 - Reader
 - 2x barrier
- Critical section
- Time access



Atomic Variables and Usage Counters

- Use of “atomic_t”
 - Explicit use of memory barriers
 - Usage counters and atomic_dec_and_test
 - Risk of cache line bouncing due to counter increments and decrements
- Effort
 - High
 - Increment
 - Decrement
 - Add
 - Low
 - Assignment
 - Store
 - Loads
 - Very high
 - Bit Operations

Example of `atomic_dec_and_test`

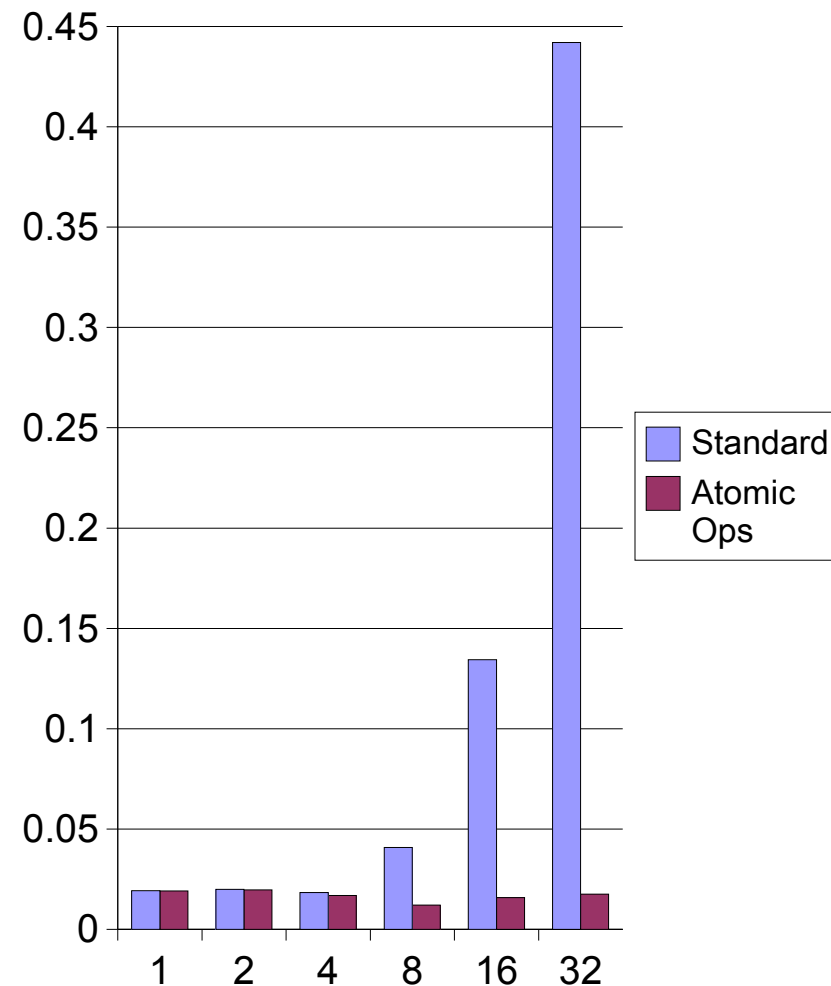
```
/*
 * Decrement the use count and release all resources for an mm.
 */
void mmput(struct mm_struct *mm)
{
    if (atomic_dec_and_test(&mm->mm_users)) {
        exit_aio(mm);
        exit_mmap(mm);
        if (!list_empty(&mm->mmlist)) {
            spin_lock(&mmlist_lock);
            list_del(&mm->mmlist);
            spin_unlock(&mmlist_lock);
        }
        put_swap_token(mm);
        mmdrop(mm);
    }
}
EXPORT_SYMBOL GPL(mmput);
```

Per CPU “Atomicity”

- Guaranteed if one processor is accessing variables reserved for its own use.
- Disabling interrupts, preemption to guaranteed non interference by interrupts or the process being moved to another processor.
- Splitting of counters per cpu to avoid atomic operations
- Counter coherency issues

Combining Techniques

- Earlier example of rcu locks and spinlocks
- Page Fault Patches
 - Page table spinlock
 - Mmap_sem
 - Limited atomic operations
- Redefining a spinlock
 - Do not modify only populate
- Severity of changing lock semantics

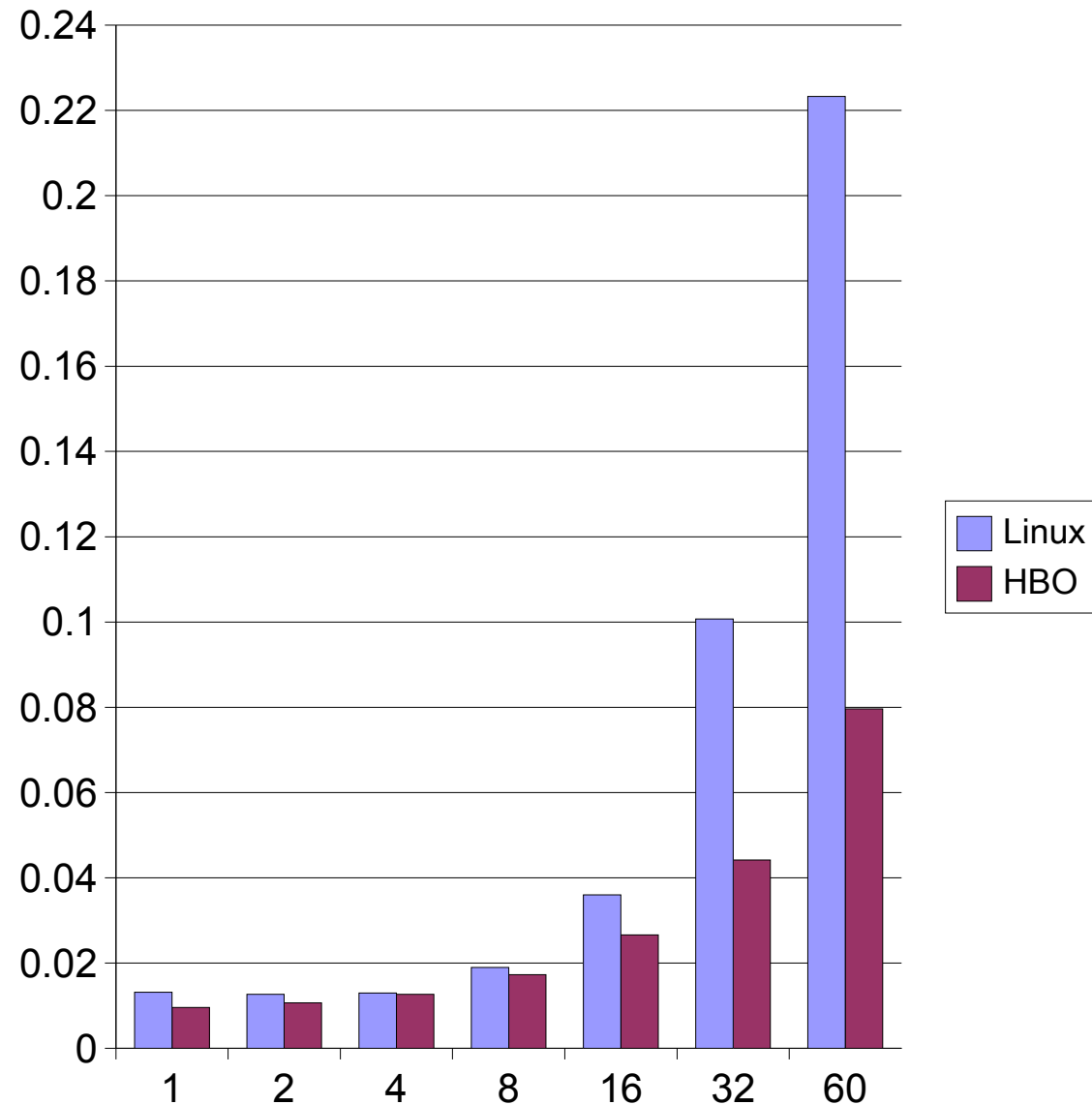


Other Locking Approaches

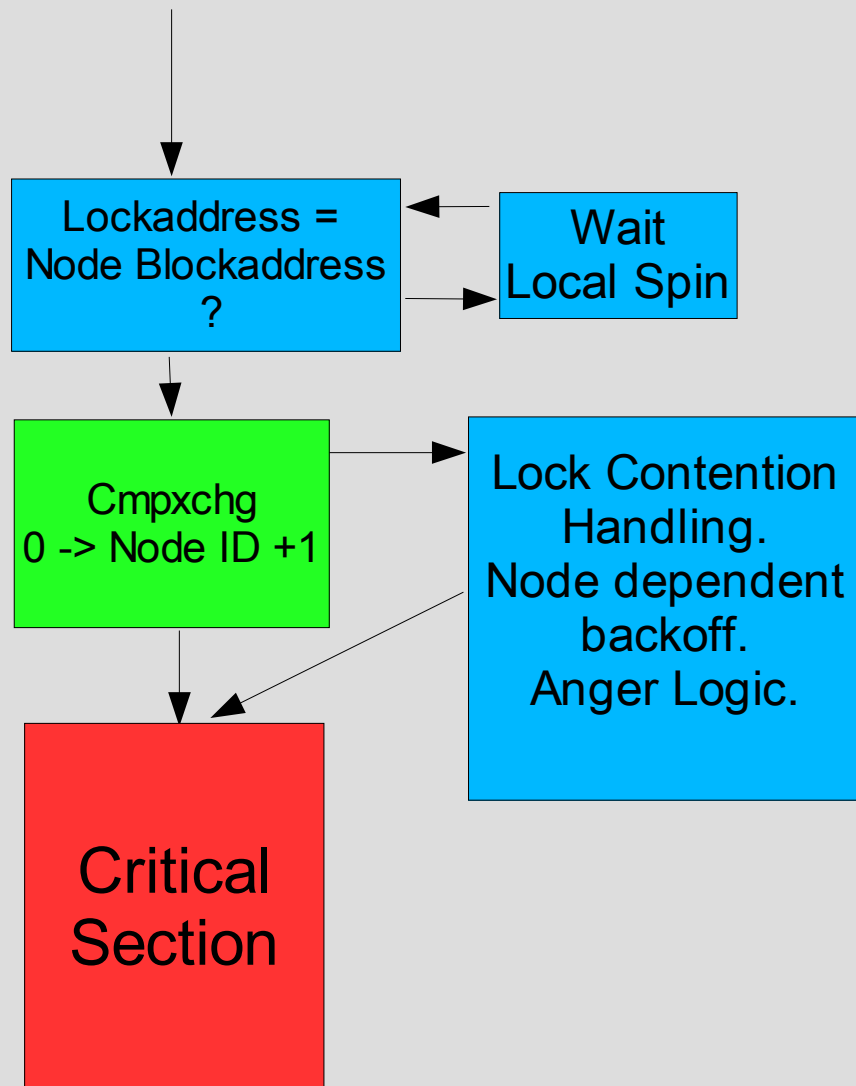
- Backoff Algorithms
 - Obvious choice
 - Simple Backoff
 - Ethernet style exponential backoff
- Queue locks
 - Access ordering
 - Slow typical combined with simple spinlock
 - Fairness addressed
 - MCS
 - John Stultz MCS Queue implementation for Linux
- Locking based on Hardware features
 - Bypass cache coherency protocol

Hierarchical BackOff Locks

- HBO
 - NUMA aware backoff
 - Limit off node contention
 - Starvation and Anger Levels
- Disadvantages
 - Additional load operation
 - Complexity of contention handling



HBO Details



- Contention handling
 - Backoff
 - On node -> 4 microsecond backoff
 - Off node -> 7 microseconds
 - 50% backoff increase on failure
 - Off node
 - Set blockaddress
 - Anger Level
 - After 50 retries set remote blockaddress