Generic Mutex Subsystem of Linux Kernel

Keyan Chen(kc32), Zhiyuan Tang(zt6)

8 pages

Abstract

- Motivation: Both of us are computer science undergraduate students and we are all very interested in parallel programming. While we were taking COMP322 last year, we found out, astonishingly, that the choice of different locking mechanisms such as mutex, shared-locks, exclusive-locks, and semaphores can have a a significant influence on the overall performance of the program. Therefore, in order to have a better understanding of multi-thread programming and in order to gain more knowledge into those locking mechanisms that we used before, we decided to research mutex subsystem of linux kernel.
- Problem Statements: Our main goal of the research is to understand the implementation details and design highlights of linux kernel mutex subsystem, as well as its advantages over other locking mechanisms and its predecessors. Lastly, as suggested by Dr. Zhong, we also researched linux implementation of atomic operations.
- Conclusion: In short, we should use mutex whenever possible instead of semaphore because of the various advantages of mutex over semaphore. In addition, the idea of optimistic spinning and the design of MCS lock also inspire us of a lot of possible optimization that we can implement in the future in our multi-thread programs. More detailed conclusions can be found in our Conclusion section.

1. Breif History and Background

In the linux kernel, mutexes refer to a particular locking primitive that enforces serialization on shared memory systems, and not only to the generic term referring to mutual exclusion found in academia or similar theoretical textbooks. Before 2006, when developers wanted to gain mutual exclusion among their shared memory systems, they would use binary semaphores, which are sleeping locks. However, Mutexes were introduced in 2006 as an alternative to binary semaphores and this new data structure provided a number of advantages, including simpler interfaces, and at that time smaller code.

1.1 Why Mutex?

Why do we need a new mutex subsystem? And what's wrong with semaphores?

1. 'struct mutex' is smaller.
   (a) On x86, 'struct semaphore' is 20 bytes, 'struct mutex' is 16 bytes. A smaller structure size means less RAM footprint, and better CPU-cache utilization
2. Mutex can result in tighter code
   (a) On x86 we can get the following .text sizes when switching all mutex-alike semaphores in the kernel to the mutex subsystem

```
   text    data     bss     dec     hex filename
3280380  868188  396860 4545428  455b94 vmlinux-semaphore
3255329  865296  396732 4517357  44eded vmlinux-mutex
```

   (b) That's 25021 bytes of code saves, or a 0.76% win off the hottest codes paths of the kernel
   (c) Smaller code means better in-cache footprint, which is one of the major optimization goals in the linux kernel when people were proposing the addition of mutex subsystem in 2006.
3. The mutex subsystem is faster and has superior scalability for contented workloads.
   (a) On a 8-way x86 system, running a mutex based kernel and testing create+unlink+close(of separate, per-task

files) in /tmp with 16 parallel tasks, the average number of ops/sec is

```
Semaphores:                          Mutexes:

$ ./test-mutex V 16 10               $ ./test-mutex V 16 10
8 CPUs, running 16 tasks.            8 CPUs, running 16 tasks.
checking VFS performance.            checking VFS performance.
avg loops/sec:      34713            avg loops/sec:      84153
CPU utilization:    63%              CPU utilization:    22%
```

(b) In this workload, mutex based kernel was **2.4 times** faster than the semaphore based kernel, and it also had **2.8 times** less CPU utilization.

1.2 Design and Implementation details

Mutex is represented by struct mutex, defined in include/linux/mutex.h and implemented in kernel/locking/mutex.c. The mutex uses a three state atomic counter to represent the different possible transitions that can occur during the lifetime of a mutex:

- 1: unlocked
- 0: locked, no waiters
- negative: locked, with potential waiters

In its most basic form it also includes a wait-queue and a spinlock that serializes access to it. When acquring a mutex, there are three possible paths that can be taken, depending on the state of lock:

1. Fastpath: tries to atomically acquire the lock by decrementing the counter. If it was already taken by another task it goes to the next possible path. This logic is architecture specific.
2. Mid-path: aka optimistic spinning. It tries to spin for acquisition while the lock owner is running and there are no other tasks ready to run that have higher priority. The rationale is that if the lock owner is running, it is likely to release the lock soon. The mutex spinners are queued up using MCS lock so that only one spinner can compete for the mutex. The MCS lock is a simple spinlock with the desirable properties of being fair and with each cpu trying to acquire the lock spinning on a local variable. It avoids expensive cache-line bouncing that common test-and-set spinlock implementations incur. And MCS-like lock is specially tailored for optimistic spinning for sleeping lock implementation.
3. Slowpath: last resort, if the lock is still unable to be acquired, the task is added to the wait-queue and sleeps until woken up by the unlock path. Under normal circumstances it blocks as TASK_UNINTERRUPTIBLE.

2. Interesting Findings

2.1 Unique Optimistic Spinning in Linux Mutex Subsystem

While formally kernel mutexes are sleepable locks, it is the Mid-path(aka optimistic spinning) that makes the mutex subsystem now more practically a hybrid type. By simply not interrupting a task and busy-waiting for a few cycles instead of immediately sleeping, the performance of this lock has been seen to significantly improve a number of workloads.

```
static bool mutex_optimistic_spin(struct mutex *lock,
                                  struct ww_acquire_ctx *ww_ctx,
                                  const bool use_ww_ctx, const bool waiter)
{
        struct task_struct *task = current;
        if (!waiter) {
                if (!mutex_can_spin_on_owner(lock))
                        goto fail;
                if (!osq_lock(&lock->osq))
                        goto fail;
        }

        for (;;) {
                struct task_struct *owner;

                if (use_ww_ctx && ww_ctx->acquired > 0) {
                        struct ww_mutex *ww;
                        ww = container_of(lock, struct ww_mutex, base);
                        if (READ_ONCE(ww->ctx))
                                goto fail_unlock;
                }

                /*
                 * If there's an owner, wait for it to either
                 * release the lock or go to sleep.
                 */
                owner = __mutex_owner(lock);
                if (owner) {
                        if (waiter && owner == task) {
                                smp_mb(); /* ACQUIRE */
                                break;
                        }

                        if (!mutex_spin_on_owner(lock, owner))
                                goto fail_unlock;
                }

                /* Try to acquire the mutex if it is unlocked. */
                if (__mutex_trylock(lock, waiter))
                        break;
                cpu_relax();
        }

        if (!waiter)
                osq_unlock(&lock->osq);

        return true;


fail_unlock:
        if (!waiter)
                osq_unlock(&lock->osq);

fail:
        if (need_resched()) {
                __set_current_state(TASK_RUNNING);
                schedule_preempt_disabled();
        }

        return false;
}
```

Figure 1. Use of MCS and the Optimistic Spinning routine in the source code

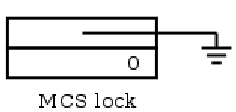
2.2 Design and Use of MCS locks in linux kernel

In mutex subsystem, mutex spinners are queued up using MCS locks as talked about above. The design of MCS lock and the difference between MCS lock and ordinary spinlock is also a very interesting design decision in linux kernel.
The concept of a spinlock is simple and straight-forward. When a thread wants to acquire the lock it will attempt to set the lock bit of that spinlock with an atomic compare-and-swap(CAS) instruction and repeatedly spin there in the lock can not be acquired during one CAS.
However, spinlocks have some fundamental problems. One of those is that every attemp to acquire a lock requires moving the cache line containing that lock to the local CPU. This cache-line bouncing can be extremely bad to performance for contended locks. Therefore, developers had been working on reducing the cache contention of spinlocks and thus, MCS locks were introduced by Tim Chen to solve this problem.
The structure of MCS lock is defined in struct mcs_spinlock

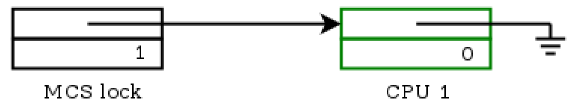
```
struct mcs_spinlock {
    struct mcs_spinlock *next;
    int locked; /* 1 if lock acquired */
};
```

Here we will go step-by-step to demonstrate how a MCS lock works and thus, we can see the difference between MCS lock and ordinary spinlock, as well as why MCS lock is guaranteed to provide a FIFO ordering on unlocking waiting threads.

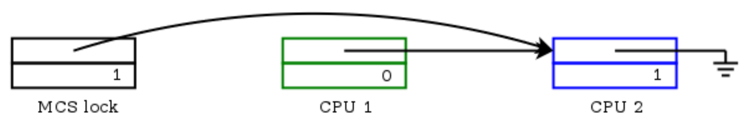
- From mcs_spinlock we can visualize an unlocked MCS lock as the following:



- Now when a new CPU thread is going to acquire the MCS lock, it will instantiate a mcs_spinlock structure on its own thread and use an unconditional atomic exchange operation to try to store the address of its own mcs_spinlock in the next field of the main MCS lock. The atomic exchange will return the previous value of the main MCS lock's next field. In this case when the previous next value of null pointer is returned back to the CPU thread acquiring the lock, it will know that is successfully acquires the lock. At this point the system can be visualized in this figure:



- Most interesting thing comes to our view when a second CPU thread is going to acquire the same MCS lock. If there's a second CPU thread trying to acquire the MCS lock while the lock hasn't been unlocked yet, the thread will still try the same locking routine as specified above. And then the returned mcs_spinlock address from the atomic swap will be a pointer to the first CPU thread's mcs_spinlock structure. In this case the second CPU will know that it doesn't acquire the MCS lock and will thus, store a pointer to its own mcs_spinlock structure in the next field of the first CPU thread's mcs_spinlock structure. The system at this point can be visualized in this figure:



- Once this assignment is done, the second CPU thread will spin on the locked value in its own mcs_spinlock structure rather than the locked value in the main MCS lock's structure. Therefore, the second CPU thread's spinning will be entirely CPU-local. As more CPU threads are joining this waiting queue, each thread will be spinning on the locked value of its local mcs_spinlock structure and form a chain of waiting queue, which by its structure, guarantees a FIFO unlock ordering.
- Finally when the first CPU thread is going to unlock the MCS lock, it will first try to do a compare-and-swap(CAS) operation on the main MCS lock's next field to set it to null pointer under the assumption that the next field still points to itself. In that CAS operation fails, the first thread will know that there are other waiting threads and thus, find the mcs_spinlock structure of the second thread and change its locked value in order to stop second thread from spinning and unlock it. The system at this point can be visualized in this figure:

In conclusion, an MCS lock is more complicated than a ordinary spinlock. But the added complexity removes much of the cache-line bouncing in ordinary spinlock and ensures unlocking fairness. Therefore, mutex subsystem in linux kernel chooses to use MCS locks to queue up mutex spinners.

2.3 Bug in an obviously correct reference count code pattern

This bug comes from a kernel crash reported in July 2013. And this bug report had not been resolved until December 2013. And people found out surprisingly that an ”obviously correct” reference count code pattern turns out to have potential data race problems that can lead to dangerous bugs. And understanding this bug requires a deep understanding about linux kernel mutex subsystem, especially its unique optimistic spinning phase. Before going deep into the analysis of the bug, let’s first see a piece of code, which manipulates a structure called ”s” and this structure is protected by a mutex embedded with it.

```
int free = 0;

mutex_lock(&s->lock);
if (--s->refcount == 0) {
        free = 1;
}

mutex_unlock(&s->lock);
if (free) {
        kfree(s);
}
```

From a first look, this piece of code just ”works”. It simply locks s, decrements reference counter to s, detects whether we can free s, and then unlock s. However, because of the fact that current implementation of mutex has an optimistic spinning phase while acquiring the lock, this piece of code is no longer data race free.
The structure mutex has an atomic counter and a spinlock. When the lock is free and one thread is going to acquire lock, it will atomically decrement the counter to 0 and continue, which is known as the fast path as specified in section 1.2.
And when the thread is going to unlock mutex, it will atomically increment counter to 1 if counter is 0, which means that there are no waiting threads on this mutex currently. If the current value of counter is negative, the mutex_unlock() routine need to also wake up the first waiting thread on mutex. In code the lifecycle of a mutex can be shown as below.

```
spin_lock(&lock->wait_lock);
atomic_set(&lock->count, 1);
// wake up first waiting thread
wake_up_process();
spin_unlock(&lock->wait_lock);
```

At this point the bug gradually becomes clear to us. Because of the optimistic spinning phase, a newly coming thread which is currently doing optimistic spinning will immediately take the lock once it sees the effect of

```
atomic_set(&lock->count, 1);
```

Therefore, when the original owner of lock is calling

```
wake_up_process();
```

to wake up the next waiting thread, a newly coming thread already believes that it takes the lock and if this newly coming thread quickly frees the data structure containing the lock, the final

```
spin_unlock(&lock->wait_lock);
```

will be applied to already freed memory space, and thus causing all sorts of problems.
Therefore, Linus Torvalds, the people who detected this concurrency bug, concludes:

In other words, it’s unsafe to protect reference counts inside objects with anything but spinlocks and/or atomic refcounts. Or you have to have the lock ”outside” the object you’re protecting(which is often what you want for other reasons anyway, notably lookup)

3. Linux Implementation of Atomic Operations

3.1 Why we include implementation of atomic operations as part of our research of mutex subsystem

”Only knowing how to use computer instructions without knowing how those instructions are implemented under the hood” is a pretty common description for superficial computer science students from others especially Electrical Engineering people. And during our initial research into mutex subsystem, we made the same mistake of becoming such a superficial computer science students.
Fortunately, thanks for the timely suggestion from Dr. Zhong during our project presentation, we quickly found out that our research about mutex subsystem was too constrained on a software level without going deep into the implementations of locks and atomic operations on a hardware level. Therefore, we spent a lot of time in the later parts of our research journey into understanding the actual implementation of atomic operations.

3.2 Intro

Linux provides an abstraction of atomic operations. However, the implementation details vary from different architectures.

As what we discussed in the final presentation, all atomic operations can be performed by disabling interrupts, emulating atomic operations, and re-enabling interrupts. The advantage of this approach is that the logic is simple and the operating system does not need special instructions from the hardware. Thus, this approach is indeed used as the most generic implementation of atomic operations. The details can be found in Linux/include/asm-generic/atomic.h and the code is posted below.

```
#include <linux/irqflags.h>

#define ATOMIC_OP(op, c_op)
static inline void atomic_##op(int i, atomic_t *v)
{
        unsigned long flags;

        raw_local_irq_save(flags);
        v->counter = v->counter c_op i;
        raw_local_irq_restore(flags);
}

#define ATOMIC_OP_RETURN(op, c_op)
static inline int atomic_##op##_return(int i, atomic_t *v)
{
        unsigned long flags;
        int ret;

        raw_local_irq_save(flags);
        ret = (v->counter = v->counter c_op i);
        raw_local_irq_restore(flags);

        return ret;
}
```

Figure 2. Generic Atomic Operations in Linux/include/asm-generic/atomic.h

However, the approach we mentioned above does not utilize the atomic operations supported by the hardware. Thus, it is generally slower. For different architectures, Linux provides different implementation of atomic operations. Our discussion will focus on two most common architectures: x86 and ARM.

3.3 Atomic Operations on ARM Architecture

On Arm architecture, there are existing atomic instructions. Thus, Linux utilized them to perform atomic operations directly. On architectures with ARMv5 or earlier version, Linux utilizes swp operations to implement atomic operations.

swp Rd Rm Rn

- Rn contains an address space in memory.
- Data from memory is loaded into Rd .
- Content of Rm is saved in memory.
- If Rd == Rm, swap content of register and memory.

There is a special case, on StrongARM CPU, the implementation of swp operation is itself bogus, since it totally bypasses the cache. Linux developer decided to use the basic method to solve this problem: disable interrupts, emulate atomic operations, re-enable interrupts.

On architecture with ARMv6 or later version, Linux relies on LDREX and STREX instructions to implement atomic operations.

LDREX Rd, [Rn]

- RD is the destination register. After completion, it contains the data loaded from memory.
- Rn is the register holding the memory.

LDREX instruction guarantees that the current processor has the exclusive access to the physical address loaded.

STREX Rd, Rm, [Rn]

- RD is the destination register. After completion, it contains either 0 (if the instruction succeed), or 1 (if the instruction is locked out). - Rm is the source register holding the data to store to memory. - Rn is the register holding the memory address.

STREX performs an atomic store to the memory.

The detailed usages of those instructions we mentioned above in Linux kernel is in Linux/arch/arm/include/asm/atomic.h. The code utilizing LDREX andd STREX is posted below.

```
#if __LINUX_ARM_ARCH__ >= 6

/*
 * ARMv6 UP and SMP safe atomic ops.  We use load exclusive and
 * store exclusive to ensure that these are atomic.  We may loop
 * to ensure that the update happens.
 */

#define ATOMIC_OP(op, c_op, asm_op)
static inline void atomic_##op(int i, atomic_t *v)
{
        unsigned long tmp;
        int result;

        prefetchw(&v->counter);
        __asm__ __volatile__("@ atomic_" #op "\n"
"1:     ldrex   %0, [%3]\n"
"       " #asm_op "     %0, %0, %4\n"
"       strex   %1, %0, [%3]\n"
"       teq     %1, #0\n"
"       bne     1b"
        : "=&r" (result), "=&r" (tmp), "+Qo" (v->counter)
        : "r" (&v->counter), "Ir" (i)
        : "cc");
}

#define ATOMIC_OP_RETURN(op, c_op, asm_op)
static inline int atomic_##op##_return_relaxed(int i, atomic_t *v)
{
        unsigned long tmp;
        int result;

        prefetchw(&v->counter);

        __asm__ __volatile__("@ atomic_" #op "_return\n"
"1:     ldrex   %0, [%3]\n"
"       " #asm_op "     %0, %0, %4\n"
"       strex   %1, %0, [%3]\n"
"       teq     %1, #0\n"
"       bne     1b"
        : "=&r" (result), "=&r" (tmp), "+Qo" (v->counter)
        : "r" (&v->counter), "Ir" (i)
        : "cc");

        return result;
}
```

Figure 3. Atomic Operations for ARM Architecture in Linux/arch/arm/include/asm/atomic.h

3.4 Atomic Operations on x86 Architecture

In x86 architecture, Linux utilizes CMPXCHG instruction to implement atomic operations.
CMPXCHG Op1 Op2
- Op1 is the destination operand. The value in this operand is compared with the AL, AX, or EAX register (depending on the size of the operand). If they are equal, the value in the source operand is loaded into the destination operand. Otherwise, the value in the destination operand is loaded into the AL, AX, or EAX register.
- Op2 is the source operand.

When CMPXCHG instruction is used with LOCK prefix, the instruction is done atomically. Linux utilizes this specific instruction in implementing atomic operations on x86 architecture.
The detailed usage of CMPXCHG instructions is in Linux/arch/x86/include/asm/atomic.h. The source code is posted below. Please note that the code posted is in Linux/arch/x86/include/asm/cmpxchg.h instead of atomic.h. The reason is that atomic.h utilize cmpxchg.h to implement atomic operations.

```
/*
 * Atomic compare and exchange.  Compare OLD with MEM, if identical,
 * store NEW in MEM.  Return the initial value in MEM.  Success is
 * indicated by comparing RETURN with OLD.
 */
#define __raw_cmpxchg(ptr, old, new, size, lock)                      \
({                                                                    \
        __typeof__(*(ptr)) __ret;                                     \
        __typeof__(*(ptr)) __old = (old);                             \
        __typeof__(*(ptr)) __new = (new);                             \
        switch (size) {                                               \
        case __X86_CASE_B:                                            \
        {                                                             \
                volatile u8 *__ptr = (volatile u8 *)(ptr);            \
                asm volatile(lock "cmpxchgb %2,%1"                    \
                             : "=a" (__ret), "+m" (*__ptr)            \
                             : "q" (__new), "" (__old)               \
                             : "memory");                             \
                break;                                                \
        }                                                             \
        case __X86_CASE_W:                                            \
        {                                                             \
                volatile u16 *__ptr = (volatile u16 *)(ptr);          \
                asm volatile(lock "cmpxchgw %2,%1"                    \
                             : "=a" (__ret), "+m" (*__ptr)            \
                             : "r" (__new), "" (__old)               \
                             : "memory");                             \
                break;                                                \
        }                                                             \
        case __X86_CASE_L:                                            \
        {                                                             \
                volatile u32 *__ptr = (volatile u32 *)(ptr);          \
                asm volatile(lock "cmpxchgl %2,%1"                    \
                             : "=a" (__ret), "+m" (*__ptr)            \
                             : "r" (__new), "" (__old)               \
                             : "memory");                             \
                break;                                                \
        }                                                             \
        case __X86_CASE_Q:                                            \
        {                                                             \
                volatile u64 *__ptr = (volatile u64 *)(ptr);          \
                asm volatile(lock "cmpxchgq %2,%1"                    \
                             : "=a" (__ret), "+m" (*__ptr)            \
                             : "r" (__new), "" (__old)               \
                             : "memory");                             \
                break;                                                \
        }                                                             \
        default:                                                      \
                __cmpxchg_wrong_size();                               \
        }                                                             \
        __ret;                                                        \
})
```

Figure 4. Atomic Operations for x86 Architecture in Linux/arch/x86/include/asm/cmpxchg.h.h

3.5 Summary

Atomic operation itself is a good example of the functionality of an operating system. From a user's perspective, atomic operations are always supported, while the implementation details are hidden. From hardware's perspective, an atomic operation is simply a sequence of instructions. Operating system successfully decouples the user interface and hardware implementation details. Moreover, atomic operation is also a good example of the cooperation between software and hardware. With support from hardware, atomic operation can be implemented efficiently.

4. Approaches and Tools

The analysis of Linux source code is done on the website of Linux Cross Reference (http://lxr.free-electrons.com/source/). We analyzed the latest version (4.10) and did not analyze older version of Linux because our focus in this project is not the development history of mutex subsystem but the actual implementation and design highlights of mutex subsystem.

4.1 For Implementation and Design Highlights of Linux Mutex Subsystem

We first read the design doc of Linux Mutex Subsystem (Linux/Documentation/mutex-design.txt). After understanding the design of mutex subsystem, we looked thoroughly into the source code for mutex to figure out how the design is implemented in Linux kernel source code.
After all those steps, we had had a good understanding of the implementation of mutex subsystem and found ourselves very interested in the unique optimistic spinning phase of mutex locking routine. Therefore, we started to look into the optimistic spinning phase and the special MCS locks used in optimistic spinning.

4.2 For Linux Implementation of Atomic Operations

Besides linux source code, we rely on http://infocenter.arm.com/help/index.jsp to check the ARM instructions and http://x86.renejeschke.de to check x86 instructions.

5. Conclusion

Firstly, it's obvious that we should prefer to use mutex over semaphore in the future while we are developing multi-thread programs because of the various advantages of struct mutex.

Secondly, we found ourselves really inspired by the optimistic spinning phase of Linux mutex locking routine because the same idea of a few cycles of busy waiting before putting a thread to sleep can be used in many other concurrency control scenarios. For example, in our own multi-thread programs if we are entering a particular critical section but fails to acquire the lock at the current point, we can have a busy loop to try to acquire the lock several times before we block the thread by the actual lock function call which will block the current thread.