atomic_ops.txt

Semantics and Behavior of Atomic and Bitmask Operations

David S. Miller

This document is intended to serve as a guide to Linux port maintainers on how to implement atomic counter, bitops, and spinlock interfaces properly.

The atomic_t type should be defined as a signed integer. Also, it should be made opaque such that any kind of cast to a normal C integer type will fail. Something like the following should suffice:

```
        typedef struct { volatile int counter; } atomic_t;
```

Historically, counter has been declared volatile. This is now discouraged. See Documentation/volatile-considered-harmful.txt for the complete rationale.

local_t is very similar to atomic_t. If the counter is per CPU and only updated by one CPU, local_t is probably more appropriate. Please see Documentation/local_ops.txt for the semantics of local_t.

The first operations to implement for atomic_t's are the initializers and plain reads.

```
        #define ATOMIC_INIT(i)          { (i) }
        #define atomic_set(v, i)        ((v)->counter = (i))
```

The first macro is used in definitions, such as:

```
static atomic_t my_counter = ATOMIC_INIT(1);
```

The initializer is atomic in that the return values of the atomic operations are guaranteed to be correct reflecting the initialized value if the initializer is used before runtime. If the initializer is used at runtime, a proper implicit or explicit read memory barrier is needed before reading the value with atomic_read from another thread.

The second interface can be used at runtime, as in:

```
        struct foo { atomic_t counter; };
        ...

        struct foo *k;

        k = kmalloc(sizeof(*k), GFP_KERNEL);
        if (!k)
                return -ENOMEM;
        atomic_set(&k->counter, 0);
```

The setting is atomic in that the return values of the atomic operations by all threads are guaranteed to be correct reflecting either the value that has been set with this operation or set with another operation. A proper implicit or explicit memory barrier is needed before the value set with the operation is guaranteed to be readable with atomic_read from another thread.

Next, we have:

```
        #define atomic_read(v)  ((v)->counter)
```

which simply reads the counter value currently visible to the calling thread.
The read is atomic in that the return value is guaranteed to be one of the
values initialized or modified with the interface operations if a proper
implicit or explicit memory barrier is used after possible runtime
initialization by any other thread and the value is modified only with the
interface operations. atomic_read does not guarantee that the runtime
initialization by any other thread is visible yet, so the user of the
interface must take care of that with a proper implicit or explicit memory
barrier.

*** WARNING: atomic_read() and atomic_set() DO NOT IMPLY BARRIERS! ***

Some architectures may choose to use the volatile keyword, barriers, or inline
assembly to guarantee some degree of immediacy for atomic_read() and
atomic_set(). This is not uniformly guaranteed, and may change in the future,
so all users of atomic_t should treat atomic_read() and atomic_set() as simple
C statements that may be reordered or optimized away entirely by the compiler
or processor, and explicitly invoke the appropriate compiler and/or memory
barrier for each use case. Failure to do so will result in code that may
suddenly break when used with different architectures or compiler
optimizations, or even changes in unrelated code which changes how the
compiler optimizes the section accessing atomic_t variables.

*** YOU HAVE BEEN WARNED! ***

Now, we move onto the atomic operation interfaces typically implemented with
the help of assembly code.

```
        void atomic_add(int i, atomic_t *v);
        void atomic_sub(int i, atomic_t *v);
        void atomic_inc(atomic_t *v);
        void atomic_dec(atomic_t *v);
```

These four routines add and subtract integral values to/from the given
atomic_t value. The first two routines pass explicit integers by
which to make the adjustment, whereas the latter two use an implicit
adjustment value of "1".

One very important aspect of these two routines is that they DO NOT
require any explicit memory barriers. They need only perform the
atomic_t counter update in an SMP safe manner.

Next, we have:

```
        int atomic_inc_return(atomic_t *v);
        int atomic_dec_return(atomic_t *v);
```

These routines add 1 and subtract 1, respectively, from the given
atomic_t and return the new counter value after the operation is
performed.

Unlike the above routines, it is required that explicit memory
barriers are performed before and after the operation. It must be
done such that all memory operations before and after the atomic
operation calls are strongly ordered with respect to the atomic
operation itself.

For example, it should behave as if a smp_mb() call existed both
before and after the atomic operation.

If the atomic instructions used in an implementation provide explicit
memory barrier semantics which satisfy the above requirements, that is
fine as well.

Let's move on:

```
	int atomic_add_return(int i, atomic_t *v);
	int atomic_sub_return(int i, atomic_t *v);
```

These behave just like atomic_{inc,dec}_return() except that an
explicit counter adjustment is given instead of the implicit "1".
This means that like atomic_{inc,dec}_return(), the memory barrier
semantics are required.

Next:

```
	int atomic_inc_and_test(atomic_t *v);
	int atomic_dec_and_test(atomic_t *v);
```

These two routines increment and decrement by 1, respectively, the
given atomic counter. They return a boolean indicating whether the
resulting counter value was zero or not.

It requires explicit memory barrier semantics around the operation as
above.

```
	int atomic_sub_and_test(int i, atomic_t *v);
```

This is identical to atomic_dec_and_test() except that an explicit
decrement is given instead of the implicit "1". It requires explicit
memory barrier semantics around the operation.

```
	int atomic_add_negative(int i, atomic_t *v);
```

The given increment is added to the given atomic counter value. A
boolean is return which indicates whether the resulting counter value
is negative. It requires explicit memory barrier semantics around the
operation.

Then:

```
	int atomic_xchg(atomic_t *v, int new);
```

This performs an atomic exchange operation on the atomic variable v, setting
the given new value. It returns the old value that the atomic variable v had
just before the operation.

```
	int atomic_cmpxchg(atomic_t *v, int old, int new);
```

This performs an atomic compare exchange operation on the atomic value v, with the given old and new values. Like all atomic_xxx operations, atomic_cmpxchg will only satisfy its atomicity semantics as long as all other accesses of *v are performed through atomic_xxx operations.

atomic_cmpxchg requires explicit memory barriers around the operation.

The semantics for atomic_cmpxchg are the same as those defined for 'cas' below.

Finally:

```
	int atomic_add_unless(atomic_t *v, int a, int u);
```

If the atomic value v is not equal to u, this function adds a to v, and returns non zero. If v is equal to u then it returns zero. This is done as an atomic operation.

atomic_add_unless requires explicit memory barriers around the operation unless it fails (returns 0).

atomic_inc_not_zero, equivalent to atomic_add_unless(v, 1, 0)

If a caller requires memory barrier semantics around an atomic_t operation which does not return a value, a set of interfaces are defined which accomplish this:

```
	void smp_mb__before_atomic_dec(void);
	void smp_mb__after_atomic_dec(void);
	void smp_mb__before_atomic_inc(void);
	void smp_mb__after_atomic_inc(void);
```

For example, smp_mb__before_atomic_dec() can be used like so:

```
	obj->dead = 1;
	smp_mb__before_atomic_dec();
	atomic_dec(&obj->ref_count);
```

It makes sure that all memory operations preceding the atomic_dec() call are strongly ordered with respect to the atomic counter operation. In the above example, it guarantees that the assignment of "1" to obj->dead will be globally visible to other cpus before the atomic counter decrement.

Without the explicit smp_mb__before_atomic_dec() call, the implementation could legally allow the atomic counter update visible to other cpus before the "obj->dead = 1;" assignment.

The other three interfaces listed are used to provide explicit ordering with respect to memory operations after an atomic_dec() call (smp_mb__after_atomic_dec()) and around atomic_inc() calls (smp_mb__{before,after}_atomic_inc()).

A missing memory barrier in the cases where they are required by the
atomic_t implementation above can have disastrous results. Here is
an example, which follows a pattern occurring frequently in the Linux
kernel. It is the use of atomic counters to implement reference
counting, and it works such that once the counter falls to zero it can
be guaranteed that no other entity can be accessing the object:

```
static void obj_list_add(struct obj *obj, struct list_head *head)
{
        obj->active = 1;
        list_add(&obj->list, head);
}

static void obj_list_del(struct obj *obj)
{
        list_del(&obj->list);
        obj->active = 0;
}

static void obj_destroy(struct obj *obj)
{
        BUG_ON(obj->active);
        kfree(obj);
}

struct obj *obj_list_peek(struct list_head *head)
{
        if (!list_empty(head)) {
                struct obj *obj;

                obj = list_entry(head->next, struct obj, list);
                atomic_inc(&obj->refcnt);
                return obj;
        }
        return NULL;
}

void obj_poke(void)
{
        struct obj *obj;

        spin_lock(&global_list_lock);
        obj = obj_list_peek(&global_list);
        spin_unlock(&global_list_lock);

        if (obj) {
                obj->ops->poke(obj);
                if (atomic_dec_and_test(&obj->refcnt))
                        obj_destroy(obj);
        }
}

void obj_timeout(struct obj *obj)
{
        spin_lock(&global_list_lock);
        obj_list_del(obj);
        spin_unlock(&global_list_lock);

        if (atomic_dec_and_test(&obj->refcnt))
                obj_destroy(obj);
}
```

(This is a simplification of the ARP queue management in the
 generic neighbour discover code of the networking. Olaf Kirch
 found a bug wrt. memory barriers in kfree_skb() that exposed
 the atomic_t memory barrier requirements quite clearly.)

Given the above scheme, it must be the case that the obj->active
update done by the obj list deletion be visible to other processors
before the atomic counter decrement is performed.

Otherwise, the counter could fall to zero, yet obj->active would still
be set, thus triggering the assertion in obj_destroy(). The error
sequence looks like this:

```
        cpu 0                           cpu 1
        obj_poke()                      obj_timeout()
        obj = obj_list_peek();
        ... gains ref to obj, refcnt=2
                                        obj_list_del(obj);
                                        obj->active = 0 ...
                                        ... visibility delayed ...
                                        atomic_dec_and_test()
                                        ... refcnt drops to 1 ...
        atomic_dec_and_test()
        ... refcount drops to 0 ...
        obj_destroy()
        BUG() triggers since obj->active
        still seen as one
                                        obj->active update visibility occurs
```

With the memory barrier semantics required of the atomic_t operations
which return values, the above sequence of memory visibility can never
happen. Specifically, in the above case the atomic_dec_and_test()
counter decrement would not become globally visible until the
obj->active update does.

As a historical note, 32-bit Sparc used to only allow usage of
24-bits of its atomic_t type. This was because it used 8 bits
as a spinlock for SMP safety. Sparc32 lacked a "compare and swap"
type instruction. However, 32-bit Sparc has since been moved over
to a "hash table of spinlocks" scheme, that allows the full 32-bit
counter to be realized. Essentially, an array of spinlocks are
indexed into based upon the address of the atomic_t being operated
on, and that lock protects the atomic operation. Parisc uses the
same scheme.

Another note is that the atomic_t operations returning values are
extremely slow on an old 386.

We will now cover the atomic bitmask operations. You will find that
their SMP and memory barrier semantics are similar in shape and scope

to the atomic_t ops above.

Native atomic bit operations are defined to operate on objects aligned
to the size of an "unsigned long" C data type, and are least of that
size. The endianness of the bits within each "unsigned long" are the
native endianness of the cpu.

```
        void set_bit(unsigned long nr, volatile unsigned long *addr);
        void clear_bit(unsigned long nr, volatile unsigned long *addr);
        void change_bit(unsigned long nr, volatile unsigned long *addr);
```

These routines set, clear, and change, respectively, the bit number
indicated by "nr" on the bit mask pointed to by "ADDR".

They must execute atomically, yet there are no implicit memory barrier
semantics required of these interfaces.

```
        int test_and_set_bit(unsigned long nr, volatile unsigned long *addr);
        int test_and_clear_bit(unsigned long nr, volatile unsigned long *addr);
        int test_and_change_bit(unsigned long nr, volatile unsigned long *addr);
```

Like the above, except that these routines return a boolean which
indicates whether the changed bit was set _BEFORE_ the atomic bit
operation.

WARNING! It is incredibly important that the value be a boolean,
ie. "0" or "1". Do not try to be fancy and save a few instructions by
declaring the above to return "long" and just returning something like
"old_val & mask" because that will not work.

For one thing, this return value gets truncated to int in many code
paths using these interfaces, so on 64-bit if the bit is set in the
upper 32-bits then testers will never see that.

One great example of where this problem crops up are the thread_info
flag operations. Routines such as test_and_set_ti_thread_flag() chop
the return value into an int. There are other places where things
like this occur as well.

These routines, like the atomic_t counter operations returning values,
require explicit memory barrier semantics around their execution. All
memory operations before the atomic bit operation call must be made
visible globally before the atomic bit operation is made visible.
Likewise, the atomic bit operation must be visible globally before any
subsequent memory operation is made visible. For example:

```
        obj->dead = 1;
        if (test_and_set_bit(0, &obj->flags))
                /* ... */;
        obj->killed = 1;
```

The implementation of test_and_set_bit() must guarantee that
"obj->dead = 1;" is visible to cpus before the atomic memory operation
done by test_and_set_bit() becomes visible. Likewise, the atomic
memory operation done by test_and_set_bit() must become visible before
"obj->killed = 1;" is visible.

Finally there is the basic operation:

```
	int test_bit(unsigned long nr, __const__ volatile unsigned long *addr);
```

Which returns a boolean indicating if bit "nr" is set in the bitmask
pointed to by "addr".

If explicit memory barriers are required around clear_bit() (which
does not return a value, and thus does not need to provide memory
barrier semantics), two interfaces are provided:

```
	void smp_mb__before_clear_bit(void);
	void smp_mb__after_clear_bit(void);
```

They are used as follows, and are akin to their atomic_t operation
brothers:

```
	/* All memory operations before this call will
	 * be globally visible before the clear_bit().
	 */
	smp_mb__before_clear_bit();
	clear_bit( ... );

	/* The clear_bit() will be visible before all
	 * subsequent memory operations.
	 */
	 smp_mb__after_clear_bit();
```

There are two special bitops with lock barrier semantics (acquire/release,
same as spinlocks). These operate in the same way as their non-_lock/unlock
postfixed variants, except that they are to provide acquire/release semantics,
respectively. This means they can be used for bit_spin_trylock and
bit_spin_unlock type operations without specifying any more barriers.

```
	int test_and_set_bit_lock(unsigned long nr, unsigned long *addr);
	void clear_bit_unlock(unsigned long nr, unsigned long *addr);
	void __clear_bit_unlock(unsigned long nr, unsigned long *addr);
```

The __clear_bit_unlock version is non-atomic, however it still implements
unlock barrier semantics. This can be useful if the lock itself is protecting
the other bits in the word.

Finally, there are non-atomic versions of the bitmask operations
provided. They are used in contexts where some other higher-level SMP
locking scheme is being used to protect the bitmask, and thus less
expensive non-atomic operations may be used in the implementation.
They have names similar to the above bitmask operation interfaces,
except that two underscores are prefixed to the interface name.

```
	void __set_bit(unsigned long nr, volatile unsigned long *addr);
	void __clear_bit(unsigned long nr, volatile unsigned long *addr);
	void __change_bit(unsigned long nr, volatile unsigned long *addr);
	int __test_and_set_bit(unsigned long nr, volatile unsigned long *addr);
	int __test_and_clear_bit(unsigned long nr, volatile unsigned long
*addr);
```

```
	int __test_and_change_bit(unsigned long nr, volatile unsigned long
*addr);
```

These non-atomic variants also do not require any special memory
barrier semantics.

The routines xchg() and cmpxchg() need the same exact memory barriers
as the atomic and bit operations returning values.

Spinlocks and rwlocks have memory barrier expectations as well.
The rule to follow is simple:

1) When acquiring a lock, the implementation must make it globally
   visible before any subsequent memory operation.

2) When releasing a lock, the implementation must make it such that
   all previous memory operations are globally visible before the
   lock release.

Which finally brings us to _atomic_dec_and_lock(). There is an
architecture-neutral version implemented in lib/dec_and_lock.c,
but most platforms will wish to optimize this in assembler.

```
	int _atomic_dec_and_lock(atomic_t *atomic, spinlock_t *lock);
```

Atomically decrement the given counter, and if will drop to zero
atomically acquire the given spinlock and perform the decrement
of the counter to zero. If it does not drop to zero, do nothing
with the spinlock.

It is actually pretty simple to get the memory barrier correct.
Simply satisfy the spinlock grab requirements, which is make
sure the spinlock operation is globally visible before any
subsequent memory operation.

We can demonstrate this operation more clearly if we define
an abstract atomic operation:

```
	long cas(long *mem, long old, long new);
```

"cas" stands for "compare and swap". It atomically:

1) Compares "old" with the value currently at "mem".
2) If they are equal, "new" is written to "mem".
3) Regardless, the current value at "mem" is returned.

As an example usage, here is what an atomic counter update
might look like:

```
void example_atomic_inc(long *counter)
{
	long old, new, ret;

	while (1) {
		old = *counter;
		new = old + 1;
```

```
                ret = cas(counter, old, new);
                if (ret == old)
                        break;
        }
}
```

Let's use cas() in order to build a pseudo-C atomic_dec_and_lock():

```
int _atomic_dec_and_lock(atomic_t *atomic, spinlock_t *lock)
{
        long old, new, ret;
        int went_to_zero;

        went_to_zero = 0;
        while (1) {
                old = atomic_read(atomic);
                new = old - 1;
                if (new == 0) {
                        went_to_zero = 1;
                        spin_lock(lock);
                }
                ret = cas(atomic, old, new);
                if (ret == old)
                        break;
                if (went_to_zero) {
                        spin_unlock(lock);
                        went_to_zero = 0;
                }
        }

        return went_to_zero;
}
```

Now, as far as memory barriers go, as long as spin_lock()
strictly orders all subsequent memory operations (including
the cas()) with respect to itself, things will be fine.

Said another way, _atomic_dec_and_lock() must guarantee that
a counter dropping to zero is never made visible before the
spinlock being acquired.

Note that this also means that for the case where the counter
is not dropping to zero, there are no memory ordering
requirements.