NAME

atomic_add, **atomic_clear**, **atomic_cmpset**, **atomic_fcmpset**, **atomic_fetchadd**, **atomic_interrupt_fence**, **atomic_load**, **atomic_readandclear**, **atomic_set**, **atomic_subtract**, **atomic_store**, **atomic_thread_fence** - atomic operations

SYNOPSIS

#include <machine/atomic.h>

void
atomic_add_[acq_|rel_]<type>(*volatile <type> *p*, *<type> v*);

void
atomic_clear_[acq_|rel_]<type>(*volatile <type> *p*, *<type> v*);

int
atomic_cmpset_[acq_|rel_]<type>(*volatile <type> *dst*, *<type> old*, *<type> new*);

int
atomic_fcmpset_[acq_|rel_]<type>(*volatile <type> *dst*, *<type> *old*, *<type> new*);

<type>
atomic_fetchadd_<type>(*volatile <type> *p*, *<type> v*);

void
atomic_interrupt_fence(*void*);

<type>
atomic_load_[acq_]<type>(*volatile <type> *p*);

<type>
atomic_readandclear_<type>(*volatile <type> *p*);

void
atomic_set_[acq_|rel_]<type>(*volatile <type> *p*, *<type> v*);

void
atomic_subtract_[acq_|rel_]<type>(*volatile <type> *p*, *<type> v*);

void
atomic_store_[rel_]<type>(*volatile <type> *p*, *<type> v*);

<type>
atomic_swap_<type>(*volatile <type> *p, <type> v*);

int
atomic_testandclear_<type>(*volatile <type> *p, u_int v*);

int
atomic_testandset_<type>(*volatile <type> *p, u_int v*);

void
atomic_thread_fence_[acq|acq_rel|rel|seq_cst](*void*);

DESCRIPTION

Atomic operations are commonly used to implement reference counts and as building blocks for synchronization primitives, such as mutexes.

All of these operations are performed *atomically* across multiple threads and in the presence of interrupts, meaning that they are performed in an indivisible manner from the perspective of concurrently running threads and interrupt handlers.

On all architectures supported by FreeBSD, ordinary loads and stores of integers in cache-coherent memory are inherently atomic if the integer is naturally aligned and its size does not exceed the processor's word size. However, such loads and stores may be elided from the program by the compiler, whereas atomic operations are always performed.

When atomic operations are performed on cache-coherent memory, all operations on the same location are totally ordered.

When an atomic load is performed on a location in cache-coherent memory, it reads the entire value that was defined by the last atomic store to each byte of the location. An atomic load will never return a value out of thin air. When an atomic store is performed on a location, no other thread or interrupt handler will observe a *torn write*, or partial modification of the location.

Except as noted below, the semantics of these operations are almost identical to the semantics of similarly named C11 atomic operations.

Types

Most atomic operations act upon a specific *type*. That type is indicated in the function name. In contrast to C11 atomic operations, FreeBSD's atomic operations are performed on ordinary integer types. The available types are:

int	unsigned integer
long	unsigned long integer
ptr	unsigned integer the size of a pointer
32	unsigned 32-bit integer
64	unsigned 64-bit integer

For example, the function to atomically add two integers is called **atomic_add_int**().

Certain architectures also provide operations for types smaller than "int".

char	unsigned character
short	unsigned short integer
8	unsigned 8-bit integer
16	unsigned 16-bit integer

These types must not be used in machine-independent code.

Acquire and Release Operations

By default, a thread's accesses to different memory locations might not be performed in *program order*, that is, the order in which the accesses appear in the source code. To optimize the program's execution, both the compiler and processor might reorder the thread's accesses. However, both ensure that their reordering of the accesses is not visible to the thread. Otherwise, the traditional memory model that is expected by single-threaded programs would be violated. Nonetheless, other threads in a multithreaded program, such as the FreeBSD kernel, might observe the reordering. Moreover, in some cases, such as the implementation of synchronization between threads, arbitrary reordering might result in the incorrect execution of the program. To constrain the reordering that both the compiler and processor might perform on a thread's accesses, a programmer can use atomic operations with *acquire* and *release* semantics.

Atomic operations on memory have up to three variants. The first, or *relaxed* variant, performs the operation without imposing any ordering constraints on accesses to other memory locations. This variant is the default. The second variant has acquire semantics, and the third variant has release semantics.

When an atomic operation has acquire semantics, the operation must have completed before any subsequent load or store (by program order) is performed. Conversely, acquire semantics do not require that prior loads or stores have completed before the atomic operation is performed. An atomic operation can only have acquire semantics if it performs a load from memory. To denote acquire semantics, the suffix "_acq" is inserted into the function name immediately prior to the "_<*type*>" suffix. For example, to subtract two integers ensuring that the subtraction is completed before any subsequent loads and

stores are performed, use **atomic_subtract_acq_int**().

When an atomic operation has release semantics, all prior loads or stores (by program order) must have completed before the operation is performed. Conversely, release semantics do not require that the atomic operation must have completed before any subsequent load or store is performed. An atomic operation can only have release semantics if it performs a store to memory. To denote release semantics, the suffix "_rel" is inserted into the function name immediately prior to the "_<*type*>" suffix. For example, to add two long integers ensuring that all prior loads and stores are completed before the addition is performed, use **atomic_add_rel_long**().

When a release operation by one thread *synchronizes with* an acquire operation by another thread, usually meaning that the acquire operation reads the value written by the release operation, then the effects of all prior stores by the releasing thread must become visible to subsequent loads by the acquiring thread. Moreover, the effects of all stores (by other threads) that were visible to the releasing thread must also become visible to the acquiring thread. These rules only apply to the synchronizing threads. Other threads might observe these stores in a different order.

In effect, atomic operations with acquire and release semantics establish one-way barriers to reordering that enable the implementations of synchronization primitives to express their ordering requirements without also imposing unnecessary ordering. For example, for a critical section guarded by a mutex, an acquire operation when the mutex is locked and a release operation when the mutex is unlocked will prevent any loads or stores from moving outside of the critical section. However, they will not prevent the compiler or processor from moving loads or stores into the critical section, which does not violate the semantics of a mutex.

Thread Fence Operations

Alternatively, a programmer can use atomic thread fence operations to constrain the reordering of accesses. In contrast to other atomic operations, fences do not, themselves, access memory.

When a fence has acquire semantics, all prior loads (by program order) must have completed before any subsequent load or store is performed. Thus, an acquire fence is a two-way barrier for load operations. To denote acquire semantics, the suffix "_acq" is appended to the function name, for example, **atomic_thread_fence_acq**().

When a fence has release semantics, all prior loads or stores (by program order) must have completed before any subsequent store operation is performed. Thus, a release fence is a two-way barrier for store operations. To denote release semantics, the suffix "_rel" is appended to the function name, for example, **atomic_thread_fence_rel**().

Although **atomic_thread_fence_acq_rel**() implements both acquire and release semantics, it is not a full

barrier. For example, a store prior to the fence (in program order) may be completed after a load subsequent to the fence. In contrast, **atomic_thread_fence_seq_cst**() implements a full barrier. Neither loads nor stores may cross this barrier in either direction.

In C11, a release fence by one thread synchronizes with an acquire fence by another thread when an atomic load that is prior to the acquire fence (by program order) reads the value written by an atomic store that is subsequent to the release fence. In constrast, in FreeBSD, because of the atomicity of ordinary, naturally aligned loads and stores, fences can also be synchronized by ordinary loads and stores. This simplifies the implementation and use of some synchronization primitives in FreeBSD.

Since neither a compiler nor a processor can foresee which (atomic) load will read the value written by an (atomic) store, the ordering constraints imposed by fences must be more restrictive than acquire loads and release stores. Essentially, this is why fences are two-way barriers.

Although fences impose more restrictive ordering than acquire loads and release stores, by separating access from ordering, they can sometimes facilitate more efficient implementations of synchronization primitives. For example, they can be used to avoid executing a memory barrier until a memory access shows that some condition is satisfied.

Interrupt Fence Operations

The **atomic_interrupt_fence**() function establishes ordering between its call location and any interrupt handler executing on the same CPU. It is modeled after the similar C11 function **atomic_signal_fence**(), and adapted for the kernel environment.

Multiple Processors

In multiprocessor systems, the atomicity of the atomic operations on memory depends on support for cache coherence in the underlying architecture. In general, cache coherence on the default memory type, VM_MEMATTR_DEFAULT, is guaranteed by all architectures that are supported by FreeBSD. For example, cache coherence is guaranteed on write-back memory by the amd64 and i386 architectures. However, on some architectures, cache coherence might not be enabled on all memory types. To determine if cache coherence is enabled for a non-default memory type, consult the architecture's documentation.

Semantics

This section describes the semantics of each operation using a C like notation.

atomic_add(*p*, *v*)

```
*p += v;
```

atomic_clear(*p*, *v*)

```
*p &= ~v;
```

atomic_cmpset(*dst*, *old*, *new*)

```
if (*dst == old) {
        *dst = new;
        return (1);
} else
        return (0);
```

Some architectures do not implement the **atomic_cmpset**() functions for the types "char", "short", "8", and "16".

atomic_fcmpset(*dst*, **old*, *new*)

On architectures implementing *Compare And Swap* operation in hardware, the functionality can be described as

```
if (*dst == *old) {
        *dst = new;
        return (1);
} else {
        *old = *dst;
        return (0);
}
```

On architectures which provide *Load Linked/Store Conditional* primitive, the write to *dst might also fail for several reasons, most important of which is a parallel write to *dst cache line by other CPU. In this case **atomic_fcmpset**() function also returns false, despite *old == *dst.

Some architectures do not implement the **atomic_fcmpset**() functions for the types "char", "short", "8", and "16".

atomic_fetchadd(*p*, *v*)

```
tmp = *p;
*p += v;
return (tmp);
```

The **atomic_fetchadd**() functions are only implemented for the types "int", "long" and "32" and do not have any variants with memory barriers at this time.

atomic_load(*p*)

```
return (*p);
```

atomic_readandclear(*p*)

```
tmp = *p;
*p = 0;
return (tmp);
```

The **atomic_readandclear**() functions are not implemented for the types "char", "short", "ptr", "8", and "16" and do not have any variants with memory barriers at this time.

atomic_set(*p*, *v*)

```
*p |= v;
```

atomic_subtract(*p*, *v*)

```
*p -= v;
```

atomic_store(*p*, *v*)

```
*p = v;
```

atomic_swap(*p*, *v*)

```
tmp = *p;
*p = v;
return (tmp);
```

The **atomic_swap**() functions are not implemented for the types "char", "short", "ptr", "8", and "16" and do not have any variants with memory barriers at this time.

atomic_testandclear(*p*, *v*)

```
bit = 1 << (v % (sizeof(*p) * NBBY));
tmp = (*p & bit) != 0;
*p &= ~bit;
return (tmp);
```

atomic_testandset(*p*, *v*)

```
bit = 1 << (v % (sizeof(*p) * NBBY));
tmp = (*p & bit) != 0;
*p |= bit;
return (tmp);
```

The **atomic_testandset**() and **atomic_testandclear**() functions are only implemented for the types "int",

"long" and "32" and do not have any variants with memory barriers at this time.

The type "64" is currently not implemented for some of the atomic operations on the arm, i386, and powerpc architectures.

RETURN VALUES

The **atomic_cmpset**() function returns the result of the compare operation. The **atomic_fcmpset**() function returns true if the operation succeeded. Otherwise it returns false and sets **old* to the found value. The **atomic_fetchadd**(), **atomic_load**(), **atomic_readandclear**(), and **atomic_swap**() functions return the value at the specified address. The **atomic_testandset**() and **atomic_testandclear**() function returns the result of the test operation.

EXAMPLES

This example uses the **atomic_cmpset_acq_ptr**() and **atomic_set_ptr**() functions to obtain a sleep mutex and handle recursion. Since the *mtx_lock* member of a *struct mtx* is a pointer, the "ptr" type is used.

```
/* Try to obtain mtx_lock once. */
#define _obtain_lock(mp, tid)                                           \
        atomic_cmpset_acq_ptr(&(mp)->mtx_lock, MTX_UNOWNED, (tid))

/* Get a sleep lock, deal with recursion inline. */
#define _get_sleep_lock(mp, tid, opts, file, line) do {                 \
        uintptr_t _tid = (uintptr_t)(tid);                              \
                                                                        \
        if (!_obtain_lock(mp, tid)) {                                   \
                if (((mp)->mtx_lock & MTX_FLAGMASK) != _tid)            \
                        _mtx_lock_sleep((mp), _tid, (opts), (file), (line));\
                else {                                                  \
                        atomic_set_ptr(&(mp)->mtx_lock, MTX_RECURSE);   \
                        (mp)->mtx_recurse++;                            \
                }                                                       \
        }                                                               \
} while (0)
```

HISTORY

The **atomic_add**(), **atomic_clear**(), **atomic_set**(), and **atomic_subtract**() operations were introduced in FreeBSD 3.0. Initially, these operations were defined on the types "char", "short", "int", and "long".

The **atomic_cmpset**(), **atomic_load_acq**(), **atomic_readandclear**(), and **atomic_store_rel**() operations were added in FreeBSD 5.0. Simultaneously, the acquire and release variants were introduced, and

support was added for operation on the types "8", "16", "32", "64", and "ptr".

The **atomic_fetchadd**() operation was added in FreeBSD 6.0.

The **atomic_swap**() and **atomic_testandset**() operations were added in FreeBSD 10.0.

The **atomic_testandclear**() and **atomic_thread_fence**() operations were added in FreeBSD 11.0.

The relaxed variants of **atomic_load**() and **atomic_store**() were added in FreeBSD 12.0.

The **atomic_interrupt_fence**() operation was added in FreeBSD 13.0.