NAME

`_umtx_op` — interface for implementation of userspace threading synchronization primitives

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <sys/types.h>
#include <sys/umtx.h>
```

int
`_umtx_op`(*void *obj*, *int op*, *u_long val*, *void *uaddr*, *void *uaddr2*);

DESCRIPTION

The `_umtx_op`() system call provides kernel support for userspace implementation of the threading synchronization primitives. The 1:1 Threading Library (libthr, −lthr) uses the syscall to implement IEEE Std 1003.1-2001 (“POSIX.1”) pthread locks, like mutexes, condition variables and so on.

STRUCTURES

The operations, performed by the `_umtx_op`() syscall, operate on userspace objects which are described by the following structures. Reserved fields and paddings are omitted. All objects require ABI-mandated alignment, but this is not currently enforced consistently on all architectures.

The following flags are defined for flag fields of all structures:

`USYNC_PROCESS_SHARED` Allow selection of the process-shared sleep queue for the thread sleep container, when the lock ownership cannot be granted immediately, and the operation must sleep. The process-shared or process-private sleep queue is selected based on the attributes of the memory mapping which contains the first byte of the structure, see `mmap`(2). Otherwise, if the flag is not specified, the process-private sleep queue is selected regardless of the memory mapping attributes, as an optimization.

See the **SLEEP QUEUES** subsection below for more details on sleep queues.

Mutex

```
struct umutex {
        volatile lwpid_t m_owner;
        uint32_t         m_flags;
        uint32_t         m_ceilings[2];
        uintptr_t        m_rb_lnk;
};
```

The `m_owner` field is the actual lock. It contains either the thread identifier of the lock owner in the locked state, or zero when the lock is unowned. The highest bit set indicates that there is contention on the lock. The constants are defined for special values:

`UMUTEX_UNOWNED` Zero, the value stored in the unowned lock.

`UMUTEX_CONTESTED` The contenion indicator.

`UMUTEX_RB_OWNERDEAD` A thread owning the robust mutex terminated. The mutex is in unlocked state.

UMUTEX_RB_NOTRECOV The robust mutex is in a non-recoverable state. It cannot be locked until reinitialized.

The m_flags field may contain the following umutex-specific flags, in addition to the common flags:

UMUTEX_PRIO_INHERIT Mutex implements *Priority Inheritance* protocol.

UMUTEX_PRIO_PROTECT Mutex implements *Priority Protection* protocol.

UMUTEX_ROBUST Mutex is robust, as described in the **ROBUST UMUTEXES** section below.

UMUTEX_NONCONSISTENT Robust mutex is in a transient non-consistent state. Not used by kernel.

In the manual page, mutexes not having UMUTEX_PRIO_INHERIT and UMUTEX_PRIO_PROTECT flags set, are called normal mutexes. Each type of mutex, i.e. normal mutexes, priority-inherited mutexes, and priority-protected mutexes, have a separate sleep queue associated with the given key.

For priority protected mutexes, the m_ceilings array contains priority ceiling values. The m_ceilings[0] is the ceiling value for the mutex, as specified by IEEE Std 1003.1-2008 (“POSIX.1”) for the *Priority Protected* mutex protocol. The m_ceilings[1] is used only for the unlock of a priority protected mutex, when unlock is done in an order other than the reversed lock order. In this case, m_ceilings[1] must contain the ceiling value for the last locked priority protected mutex, for proper priority reassignment. If, instead, the unlocking mutex was the last priority propagated mutex locked by the thread, m_ceilings[1] should contain −1. This is required because kernel does not maintain the ordered lock list.

Condition variable

```
struct ucond {
        volatile uint32_t c_has_waiters;
        uint32_t          c_flags;
        uint32_t          c_clockid;
};
```

A non-zero c_has_waiters value indicates that there are in-kernel waiters for the condition, executing the UMTX_OP_CV_WAIT request.

The c_flags field contains flags. Only the common flags, i.e. USYNC_PROCESS_SHARED, are defined for ucond.

The c_clockid member provides the clock identifier to use for timeout, when the UMTX_OP_CV_WAIT request has both the CVWAIT_CLOCKID flag and the timeout specified. Valid clock identifiers are subset of the valid clock ids for the clock_gettime(2) syscall, namely, CLOCK_REALTIME, CLOCK_VIRTUAL, CLOCK_PROF, CLOCK_MONOTONIC, CLOCK_UPTIME, CLOCK_UPTIME_PRECISE, CLOCK_UPTIME_FAST, CLOCK_REALTIME_PRECISE, CLOCK_REALTIME_FAST, CLOCK_MONOTONIC_PRECISE, CLOCK_MONOTONIC_FAST, and CLOCK_SECOND are allowed.

Reader/writer lock

```
struct urwlock {
        volatile int32_t rw_state;
        uint32_t         rw_flags;
        uint32_t         rw_blocked_readers;
        uint32_t         rw_blocked_writers;
};
```

The rw_state field is the actual lock. It contains both the flags and counter of the read locks which were granted. Names of the rw_state bits are following:

URWLOCK_WRITE_OWNER Write lock was granted.

URWLOCK_WRITE_WAITERS There are write lock waiters.

URWLOCK_READ_WAITERS There are read lock waiters.

URWLOCK_READER_COUNT(c) Returns the count of currently granted read locks.

At any given time there may be only one thread to which the writer lock is granted on the *struct rwlock*, and no threads are granted read lock. Or, at the given time, up to URWLOCK_MAX_READERS threads may be granted the read lock simultaneously, but write lock is not granted to any thread.

The following flags for the rw_flags member of *struct urwlock* are defined, in addition to the common flags:

URWLOCK_PREFER_READER If specified, immediately grant read lock requests when urwlock is already read-locked, even in presence of unsatisfied write lock requests. By default, if there is a write lock waiter, further read requests are not granted, to prevent unfair write lock waiter starvation.

The rw_blocked_readers and rw_blocked_writers members contain the count of threads which are sleeping in kernel, waiting for the associated request type to be granted. The fields are used by kernel to update the URWLOCK_READ_WAITERS and URWLOCK_WRITE_WAITERS flags of the rw_state lock after requesting thread was woken up.

Semaphore

```
struct _usem2 {
        volatile uint32_t _count;
        uint32_t          _flags;
};
```

The _count word represents a counting semaphore. A non-zero value indicates an unlocked (posted) semaphore, while zero represents the locked state. The maximal supported semaphore count is USEM_MAX_COUNT.

The _count word, besides the counter of posts (unlocks), also contains the USEM_HAS_WAITERS bit, which indicates that locked semaphore has waiting threads.

The USEM_COUNT() macro, applied to the _count word, returns the current semaphore counter, i.e. the number of posts issued on the semaphore.

The following bits for the _flags member of *struct _usem2* are defined, in addition to the common flags:

USEM_NAMED Flag is ignored by kernel.

Timeout parameter

```
struct _umtx_time {
        struct timespec _timeout;
        uint32_t        _flags;
        uint32_t        _clockid;
};
```

Several `_umtx_op`() operations allow the blocking time to be limited, failing the request if it cannot be satisfied in the specified time period. The timeout is specified by passing either the address of *struct timespec*, or its extended variant, *struct _umtx_time*, as the *uaddr2* argument of `_umtx_op`(). They are distinguished by the *uaddr* value, which must be equal to the size of the structure pointed to by *uaddr2*, casted to *uintptr_t*.

The `_timeout` member specifies the time when the timeout should occur. Legal values for clock identifier `_clockid` are shared with the *clock_id* argument to the `clock_gettime`(2) function, and use the same underlying clocks. The specified clock is used to obtain the current time value. Interval counting is always performed by the monotonic wall clock.

The `_flags` argument allows the following flags to further define the timeout behaviour:

`UMTX_ABSTIME` The `_timeout` value is the absolute time. The thread will be unblocked and the request failed when specified clock value is equal or exceeds the `_timeout.`

If the flag is absent, the timeout value is relative, that is the amount of time, measured by the monotonic wall clock from the moment of the request start.

SLEEP QUEUES

When a locking request cannot be immediately satisfied, the thread is typically put to *sleep*, which is a non-runnable state terminated by the *wake* operation. Lock operations include a *try* variant which returns an error rather than sleeping if the lock cannot be obtained. Also, `_umtx_op`() provides requests which explicitly put the thread to sleep.

Wakes need to know which threads to make runnable, so sleeping threads are grouped into containers called *sleep queues*. A sleep queue is identified by a key, which for `_umtx_op`() is defined as the physical address of some variable. Note that the *physical* address is used, which means that same variable mapped multiple times will give one key value. This mechanism enables the construction of *process-shared* locks.

A related attribute of the key is shareability. Some requests always interpret keys as private for the current process, creating sleep queues with the scope of the current process even if the memory is shared. Others either select the shareability automatically from the mapping attributes, or take additional input as the `USYNC_PROCESS_SHARED` common flag. This is done as optimization, allowing the lock scope to be limited regardless of the kind of backing memory.

Only the address of the start byte of the variable specified as key is important for determining corresponding sleep queue. The size of the variable does not matter, so e.g. sleep on the same address interpeted as *uint32_t* and *long* on a little-endian 64-bit platform would collide.

The last attribute of the key is the object type. The sleep queue to which a sleeping thread is assigned is an individual one for simple wait requests, mutexes, rwlocks, condvars and other primitives, even when the physical address of the key is same.

When waking up a limited number of threads from a given sleep queue, the highest priority threads that have been blocked for the longest on the queue are selected.

ROBUST UMUTEXES

The *robust umutexes* are provided as a substrate for a userspace library to implement POSIX robust mutexes. A robust umutex must have the `UMUTEX_ROBUST` flag set.

On thread termination, the kernel walks two lists of mutexes. The two lists head addresses must be provided by a prior call to `UMTX_OP_ROBUST_LISTS` request. The lists are singly-linked. The link to next element is provided by the `m_rb_lnk` member of the *struct umutex*.

Robust list processing is aborted if the kernel finds a mutex with any of the following conditions:

- the `UMUTEX_ROBUST` flag is not set
- not owned by the current thread, except when the mutex is pointed to by the `robust_inactive` member of the *struct umtx_robust_lists_params*, registered for the current thread
- the combination of mutex flags is invalid
- read of the umutex memory faults
- the list length limit described in `libthr`(3) is reached.

Every mutex in both lists is unlocked as if the `UMTX_OP_MUTEX_UNLOCK` request is performed on it, but instead of the `UMUTEX_UNOWNED` value, the `m_owner` field is written with the `UMUTEX_RB_OWNERDEAD` value. When a mutex in the `UMUTEX_RB_OWNERDEAD` state is locked by kernel due to the `UMTX_OP_MUTEX_TRYLOCK` and `UMTX_OP_MUTEX_LOCK` requests, the lock is granted and `EOWNERDEAD` error is returned.

Also, the kernel handles the `UMUTEX_RB_NOTRECOV` value of `the m_owner` field specially, always returning the `ENOTRECOVERABLE` error for lock attempts, without granting the lock.

OPERATIONS

The following operations, requested by the *op* argument to the function, are implemented:

`UMTX_OP_WAIT`

Wait. The arguments for the request are:

obj Pointer to a variable of type *long*.

val Current value of the **obj*.

The current value of the variable pointed to by the *obj* argument is compared with the *val*. If they are equal, the requesting thread is put to interruptible sleep until woken up or the optionally specified timeout expires.

The comparison and sleep are atomic. In other words, if another thread writes a new value to `*obj` and then issues `UMTX_OP_WAKE`, the request is guaranteed to not miss the wakeup, which might otherwise happen between comparison and blocking.

The physical address of memory where the **obj* variable is located, is used as a key to index sleeping threads.

The read of the current value of the `*obj` variable is not guarded by barriers. In particular, it is the user's duty to ensure the lock acquire and release memory semantics, if the `UMTX_OP_WAIT` and `UMTX_OP_WAKE` requests are used as a substrate for implementing a simple lock.

The request is not restartable. An unblocked signal delivered during the wait always results in sleep interruption and `EINTR` error.

Optionally, a timeout for the request may be specified.

`UMTX_OP_WAKE`

Wake the threads possibly sleeping due to `UMTX_OP_WAIT`. The arguments for the request are:

obj Pointer to a variable, used as a key to find sleeping threads.

val Up to *val* threads are woken up by this request. Specify `INT_MAX` to wake up all waiters.

`UMTX_OP_MUTEX_TRYLOCK`

Try to lock umutex. The arguments to the request are:

obj Pointer to the umutex.

Operates same as the `UMTX_OP_MUTEX_LOCK` request, but returns `EBUSY` instead of sleeping if the lock cannot be obtained immediately.

`UMTX_OP_MUTEX_LOCK`

Lock umutex. The arguments to the request are:

obj Pointer to the umutex.

Locking is performed by writing the current thread id into the `m_owner` word of the *struct umutex*. The write is atomic, preserves the `UMUTEX_CONTESTED` contention indicator, and provides the acquire barrier for lock entrance semantic.

If the lock cannot be obtained immediately because another thread owns the lock, the current thread is put to sleep, with `UMUTEX_CONTESTED` bit set before. Upon wake up, the lock conditions are re-tested.

The request adheres to the priority protection or inheritance protocol of the mutex, specified by the `UMUTEX_PRIO_PROTECT` or `UMUTEX_PRIO_INHERIT` flag, respectively.

Optionally, a timeout for the request may be specified.

A request with a timeout specified is not restartable. An unblocked signal delivered during the wait always results in sleep interruption and `EINTR` error. A request without timeout specified is always restarted after return from a signal handler.

`UMTX_OP_MUTEX_UNLOCK`

Unlock umutex. The arguments to the request are:

obj Pointer to the umutex.

Unlocks the mutex, by writing `UMUTEX_UNOWNED` (zero) value into `m_owner` word of the *struct umutex*. The write is done with a release barrier, to provide lock leave semantic.

If there are threads sleeping in the sleep queue associated with the umutex, one thread is woken up. If more than one thread sleeps in the sleep queue, the `UMUTEX_CONTESTED` bit is set together with the write of the `UMUTEX_UNOWNED` value into `m_owner`.

The request adheres to the priority protection or inheritance protocol of the mutex, specified by the `UMUTEX_PRIO_PROTECT` or `UMUTEX_PRIO_INHERIT` flag, respectively. See description of the `m_ceilings` member of the *struct umutex* structure for additional details of the request operation on the priority protected protocol mutex.

`UMTX_OP_SET_CEILING`

Set ceiling for the priority protected umutex. The arguments to the request are:

obj Pointer to the umutex.

val New ceiling value.

uaddr Address of a variable of type *uint32_t*. If not `NULL` and the update was successful, the previous ceiling value is written to the location pointed to by *uaddr*.

The request locks the umutex pointed to by the *obj* parameter, waiting for the lock if not immediately available. After the lock is obtained, the new ceiling value *val* is written to the `m_ceilings[0]` member of the *struct umutex,* after which the umutex is unlocked.

The locking does not adhere to the priority protect protocol, to conform to the POSIX requirements for the `pthread_mutex_setprioceiling`(3) interface.

`UMTX_OP_CV_WAIT`

Wait for a condition. The arguments to the request are:

obj Pointer to the *struct ucond*.

val Request flags, see below.

uaddr Pointer to the umutex.

uaddr2 Optional pointer to a *struct timespec* for timeout specification.

The request must be issued by the thread owning the mutex pointed to by the *uaddr* argument. The `c_hash_waiters` member of the *struct ucond*, pointed to by the *obj* argument, is set to an arbitrary non-zero value, after which the *uaddr* mutex is unlocked (following the appropriate protocol), and the current thread is put to sleep on the sleep queue keyed by the *obj* argument. The operations are performed atomically. It is guaranteed to not miss a wakeup from `UMTX_OP_CV_SIGNAL` or `UMTX_OP_CV_BROADCAST` sent between mutex unlock and putting the current thread on the sleep queue.

Upon wakeup, if the timeout expired and no other threads are sleeping in the same sleep queue, the `c_hash_waiters` member is cleared. After wakeup, the *uaddr* umutex is not relocked.

The following flags are defined:

`CVWAIT_ABSTIME` Timeout is absolute.

`CVWAIT_CLOCKID` Clockid is provided.

Optionally, a timeout for the request may be specified. Unlike other requests, the timeout value is specified directly by a *struct timespec*, pointed to by the *uaddr2* argument. If the `CVWAIT_CLOCKID` flag is provided, the timeout uses the clock from the `c_clockid` member of the *struct ucond*, pointed to by *obj* argument. Otherwise, `CLOCK_REALTIME` is used, regardless of the clock identifier possibly specified in the *struct _umtx_time*. If the `CVWAIT_ABSTIME` flag is supplied, the timeout specifies absolute time value, otherwise it denotes a relative time interval.

The request is not restartable. An unblocked signal delivered during the wait always results in sleep interruption and `EINTR` error.

`UMTX_OP_CV_SIGNAL` Wake up one condition waiter. The arguments to the request are:

obj Pointer to *struct ucond*.

The request wakes up at most one thread sleeping on the sleep queue keyed by the *obj* argument. If the woken up thread was the last on the sleep queue, the `c_has_waiters` member of the *struct ucond* is cleared.

`UMTX_OP_CV_BROADCAST` Wake up all condition waiters. The arguments to the request are:

obj Pointer to *struct ucond*.

The request wakes up all threads sleeping on the sleep queue keyed by the *obj* argument. The `c_has_waiters` member of the *struct ucond* is cleared.

`UMTX_OP_WAIT_UINT` Same as `UMTX_OP_WAIT`, but the type of the variable pointed to by *obj* is *u_int*, i.e. 32-bit integer.

`UMTX_OP_RW_RDLOCK` Read-lock a *struct rwlock* lock. The arguments to the request are:

obj Pointer to the lock (of type *struct rwlock*) to be read-locked.

val Additional flags to augment locking behaviour. The valid flags in the *val* argument are:

`URWLOCK_PREFER_READER`

The request obtains the read lock on the specified *struct rwlock* by incrementing the count of readers in the `rw_state` word of the structure. If the `URWLOCK_WRITE_OWNER` bit is set in the word `rw_state`, the lock was granted to a writer which has not yet relinquished its ownership. In this case the current thread is put to sleep until it makes sense to retry.

If the `URWLOCK_PREFER_READER` flag is set either in the `rw_flags` word of the structure, or in the *val* argument of the request, the presence of the threads trying to obtain the write lock on the same structure does not prevent the current thread from trying to obtain the read lock. Otherwise, if the flag is not set, and the `URWLOCK_WRITE_WAITERS` flag is set in `rw_state`, the current thread does not attempt to obtain read-lock. Instead it sets the `URWLOCK_READ_WAITERS` in the `rw_state` word and puts itself to sleep on corresponding sleep queue. Upon wakeup, the locking conditions are re-evaluated.

Optionally, a timeout for the request may be specified.

The request is not restartable. An unblocked signal delivered during the wait always results in sleep interruption and `EINTR` error.

`UMTX_OP_RW_WRLOCK`

Write-lock a *struct rwlock* lock. The arguments to the request are:

obj Pointer to the lock (of type *struct rwlock*) to be write-locked.

The request obtains a write lock on the specified *struct rwlock*, by setting the `URWLOCK_WRITE_OWNER` bit in the `rw_state` word of the structure. If there is already a write lock owner, as indicated by the `URWLOCK_WRITE_OWNER` bit being set, or there are read lock owners, as indicated by the read-lock counter, the current thread does not attempt to obtain the write-lock. Instead it sets the `URWLOCK_WRITE_WAITERS` in the `rw_state` word and puts itself to sleep on corresponding sleep queue. Upon wakeup, the locking conditions are re-evaluated.

Optionally, a timeout for the request may be specified.

The request is not restartable. An unblocked signal delivered during the wait always results in sleep interruption and `EINTR` error.

`UMTX_OP_RW_UNLOCK`

Unlock rwlock. The arguments to the request are:

obj Pointer to the lock (of type *struct rwlock*) to be unlocked.

The unlock type (read or write) is determined by the current lock state. Note that the *struct rwlock* does not save information about the identity of the thread which acquired the lock.

If there are pending writers after the unlock, and the `URWLOCK_PREFER_READER` flag is not set in the `rw_flags` member of the ∗*obj* structure, one writer is woken up, selected as described in the **SLEEP QUEUES** subsection. If the `URWLOCK_PREFER_READER` flag is set, a pending writer is woken up only if there is no pending readers.

If there are no pending writers, or, in the case that the URWLOCK_PREFER_READER flag is set, then all pending readers are woken up by unlock.

UMTX_OP_WAIT_UINT_PRIVATE
Same as UMTX_OP_WAIT_UINT, but unconditionally select the process-private sleep queue.

UMTX_OP_WAKE_PRIVATE
Same as UMTX_OP_WAKE, but unconditionally select the process-private sleep queue.

UMTX_OP_MUTEX_WAIT
Wait for mutex availability. The arguments to the request are:

obj Address of the mutex.

Similarly to the UMTX_OP_MUTEX_LOCK, put the requesting thread to sleep if the mutex lock cannot be obtained immediately. The UMUTEX_CONTESTED bit is set in the m_owner word of the mutex to indicate that there is a waiter, before the thread is added to the sleep queue. Unlike the UMTX_OP_MUTEX_LOCK request, the lock is not obtained.

The operation is not implemented for priority protected and priority inherited protocol mutexes.

Optionally, a timeout for the request may be specified.

A request with a timeout specified is not restartable. An unblocked signal delivered during the wait always results in sleep interruption and EINTR error. A request without a timeout automatically restarts if the signal disposition requested restart via the SA_RESTART flag in *struct sigaction* member sa_flags.

UMTX_OP_NWAKE_PRIVATE
Wake up a batch of sleeping threads. The arguments to the request are:

obj Pointer to the array of pointers.

val Number of elements in the array pointed to by *obj*.

For each element in the array pointed to by *obj*, wakes up all threads waiting on the *private* sleep queue with the key being the byte addressed by the array element.

UMTX_OP_MUTEX_WAKE
Check if a normal umutex is unlocked and wake up a waiter. The arguments for the request are:

obj Pointer to the umutex.

If the m_owner word of the mutex pointed to by the *obj* argument indicates unowned mutex, which has its contention indicator bit UMUTEX_CONTESTED set, clear the bit and wake up one waiter in the sleep queue associated with the byte addressed by the *obj*, if any. Only normal mutexes are supported by the request. The sleep queue is always one for a normal mutex type.

This request is deprecated in favor of `UMTX_OP_MUTEX_WAKE2` since mutexes using it cannot synchronize their own destruction. That is, the `m_owner` word has already been set to `UMUTEX_UNOWNED` when this request is made, so that another thread can lock, unlock and destroy the mutex (if no other thread uses the mutex afterwards). Clearing the `UMUTEX_CONTESTED` bit may then modify freed memory.

`UMTX_OP_MUTEX_WAKE2`

Check if a umutex is unlocked and wake up a waiter. The arguments for the request are:

obj Pointer to the umutex.

val The umutex flags.

The request does not read the `m_flags` member of the *struct umutex*; instead, the *val* argument supplies flag information, in particular, to determine the sleep queue where the waiters are found for wake up.

If the mutex is unowned, one waiter is woken up.

If the mutex memory cannot be accessed, all waiters are woken up.

If there is more than one waiter on the sleep queue, or there is only one waiter but the mutex is owned by a thread, the `UMUTEX_CONTESTED` bit is set in the `m_owner` word of the *struct umutex*.

`UMTX_OP_SEM2_WAIT`

Wait until semaphore is available. The arguments to the request are:

obj Pointer to the semaphore (of type *struct _usem2*).

Put the requesting thread onto a sleep queue if the semaphore counter is zero. If the thread is put to sleep, the `USEM_HAS_WAITERS` bit is set in the `_count` word to indicate waiters. The function returns either due to `_count` indicating the semaphore is available (non-zero count due to post), or due to a wakeup. The return does not guarantee that the semaphore is available, nor does it consume the semaphore lock on successful return.

Optionally, a timeout for the request may be specified.

A request with non-absolute timeout value is not restartable. An unblocked signal delivered during such wait results in sleep interruption and `EINTR` error.

`UMTX_OP_SEM2_WAKE`

Wake up waiters on semaphore lock. The arguments to the request are:

obj Pointer to the semaphore (of type *struct _usem2*).

The request wakes up one waiter for the semaphore lock. The function does not increment the semaphore lock count. If the `USEM_HAS_WAITERS` bit was set in the `_count` word, and the last sleeping thread was woken up, the bit is cleared.

`UMTX_OP_SHM`

Manage anonymous POSIX shared memory objects (see `shm_open`(2)), which can be attached to a byte of physical memory, mapped into the process address space. The objects are used to implement process-shared locks in `libthr`.

The *val* argument specifies the sub-request of the `UMTX_OP_SHM` request:

`UMTX_SHM_CREAT`

Creates the anonymous shared memory object, which can be looked up with the specified key *uaddr*. If the object associated with the *uaddr* key already exists, it is returned instead of creating a new object. The object's size is one page. On success, the file descriptor referencing the object is returned. The descriptor can be used for mapping the object using `mmap`(2), or for other shared memory operations.

`UMTX_SHM_LOOKUP`

Same as `UMTX_SHM_CREATE` request, but if there is no shared memory object associated with the specified key *uaddr*, an error is returned, and no new object is created.

`UMTX_SHM_DESTROY`

De-associate the shared object with the specified key *uaddr*. The object is destroyed after the last open file descriptor is closed and the last mapping for it is destroyed.

`UMTX_SHM_ALIVE`

Checks whether there is a live shared object associated with the supplied key *uaddr*. Returns zero if there is, and an error otherwise. This request is an optimization of the `UMTX_SHM_LOOKUP` request. It is cheaper when only the liveness of the associated object is asked for, since no file descriptor is installed in the process fd table on success.

The *uaddr* argument specifies the virtual address, which backing physical memory byte identity is used as a key for the anonymous shared object creation or lookup.

`UMTX_OP_ROBUST_LISTS`

Register the list heads for the current thread's robust mutex lists. The arguments to the request are:

val

Size of the structure passed in the *uaddr* argument.

uaddr Pointer to the structure of type *struct umtx_robust_lists_params*.

The structure is defined as

```
struct umtx_robust_lists_params {
        uintptr_t       robust_list_offset;
        uintptr_t       robust_priv_list_offset;
        uintptr_t       robust_inact_offset;
};
```

The `robust_list_offset` member contains address of the first element in the list of locked robust shared mutexes. The `robust_priv_list_offset` member contains address of the first element in the list of locked robust private mutexes. The private and shared robust locked lists are split to allow fast termination of the shared list on fork, in the child.

The `robust_inact_offset` contains a pointer to the mutex which might be locked in nearby future, or might have been just unlocked. It is typically set by the lock or unlock mutex implementation code around the whole operation, since lists can be only changed race-free when the thread owns the mutex. The kernel inspects the `robust_inact_offset` in addition to walking the shared and private lists. Also, the mutex pointed to by `robust_inact_offset` is handled more loosely at the thread termination time, than other mutexes on the list. That mutex is allowed to be not owned by the current thread, in which case list processing is continued. See **ROBUST UMUTEXES** subsection for details.

RETURN VALUES

If successful, all requests, except `UMTX_SHM_CREAT` and `UMTX_SHM_LOOKUP` sub-requests of the `UMTX_OP_SHM` request, will return zero. The `UMTX_SHM_CREAT` and `UMTX_SHM_LOOKUP` return a shared memory file descriptor on success. On error −1 is returned, and the *errno* variable is set to indicate the error.

ERRORS

The `_umtx_op`() operations will return the following errors:

[`EFAULT`] One of the arguments point to invalid memory.

[`EINVAL`] The clock identifier, specified for the *struct _umtx_time* timeout parameter, or in the `c_clockid` member of *struct ucond,* is invalid.

[`EINVAL`] The type of the mutex, encoded by the `m_flags` member of *struct umutex*, is invalid.

[`EINVAL`] The `m_owner` member of the *struct umutex* has changed the lock owner thread identifier during unlock.

[`EINVAL`] The `timeout.tv_sec` or `timeout.tv_nsec` member of *struct _umtx_time* is less than zero, or `timeout.tv_nsec` is greater than 1000000000.

[EINVAL] The *op* argument specifies invalid operation.

[EINVAL] The *uaddr* argument for the UMTX_OP_SHM request specifies invalid operation.

[EINVAL] The UMTX_OP_SET_CEILING request specifies non priority protected mutex.

[EINVAL] The new ceiling value for the UMTX_OP_SET_CEILING request, or one or more of the values read from the m_ceilings array during lock or unlock operations, is greater than RTP_PRIO_MAX.

[EPERM] Unlock attempted on an object not owned by the current thread.

[EOWNERDEAD] The lock was requested on an umutex where the m_owner field was set to the UMUTEX_RB_OWNERDEAD value, indicating terminated robust mutex. The lock was granted to the caller, so this error in fact indicates success with additional conditions.

[ENOTRECOVERABLE] The lock was requested on an umutex which m_owner field is equal to the UMUTEX_RB_NOTRECOV value, indicating abandoned robust mutex after termination. The lock was not granted to the caller.

[ENOTTY] The shared memory object, associated with the address passed to the UMTX_SHM_ALIVE sub-request of UMTX_OP_SHM request, was destroyed.

[ESRCH] For the UMTX_SHM_LOOKUP, UMTX_SHM_DESTROY, and UMTX_SHM_ALIVE subrequests of the UMTX_OP_SHM request, there is no shared memory object associated with the provided key.

[ENOMEM] The UMTX_SHM_CREAT sub-request of the UMTX_OP_SHM request cannot be satisfied, because allocation of the shared memory object would exceed the RLIMIT_UMTXP resource limit, see setrlimit(2).

[EAGAIN] The maximum number of readers (URWLOCK_MAX_READERS) were already granted ownership of the given *struct rwlock* for read.

[EBUSY] A try mutex lock operation was not able to obtain the lock.

[ETIMEDOUT] The request specified a timeout in the *uaddr* and *uaddr2* arguments, and timed out before obtaining the lock or being woken up.

[EINTR] A signal was delivered during wait, for a non-restartable operation. Operations with timeouts are typically non-restartable, but timeouts specified in absolute time may be restartable.

[ERESTART] A signal was delivered during wait, for a restartable operation. Mutex lock requests without timeout specified are restartable. The error is not returned to userspace code since restart is handled by usual adjustment of the instruction counter.

BUGS

A window between a unlocking robust mutex and resetting the pointer in the robust_inact_offset member of the registered *struct umtx_robust_lists_params* allows another thread to destroy the mutex, thus making the kernel inspect freed or reused memory. The libthr implementation is only vulnerable to this race when operating on a shared mutex. A possible fix for the current implementation is to strengthen the checks for shared mutexes before terminating them, in particular, verifying that the mutex memory is mapped from a shared memory object allocated by the UMTX_OP_SHM request. This is not done because it is believed that the race is adequately covered by other consistency checks, while adding the check would prevent alternative implementations of libpthread.

SEE ALSO

clock_gettime(2), mmap(2), shm_open(2), setrlimit(2), sigaction(2), thr_exit(2), thr_kill(2), thr_kill2(2), thr_new(2), thr_self(2), thr_set_name(2), signal(3)

STANDARDS

The _umtx_op() system call is non-standard and is used by the 1:1 Threading Library (libthr, −lthr) to implement IEEE Std 1003.1-2001 (“POSIX.1”) pthread(3) functionality.