NAME

futex − fast user-space locking

SYNOPSIS

#include <linux/futex.h>
#include <sys/time.h>

int futex(int **uaddr*, **int** *futex_op*, **int** *val*,
    const struct timespec **timeout*, /* or: **uint32_t** *val2* */
    int **uaddr2*, **int** *val3***);**

Note: There is no glibc wrapper for this system call; see NOTES.

DESCRIPTION

The **futex**() system call provides a method for waiting until a certain condition becomes true. It is typically used as a blocking construct in the context of shared-memory synchronization. When using futexes, the majority of the synchronization operations are performed in user space. A user-space program employs the **futex**() system call only when it is likely that the program has to block for a longer time until the condition becomes true. Other **futex**() operations can be used to wake any processes or threads waiting for a particular condition.

A futex is a 32-bit value—referred to below as a *futex word*—whose address is supplied to the **futex**() system call. (Futexes are 32 bits in size on all platforms, including 64-bit systems.) All futex operations are governed by this value. In order to share a futex between processes, the futex is placed in a region of shared memory, created using (for example) **mmap**(2) or **shmat**(2). (Thus, the futex word may have different virtual addresses in different processes, but these addresses all refer to the same location in physical memory.) In a multithreaded program, it is sufficient to place the futex word in a global variable shared by all threads.

When executing a futex operation that requests to block a thread, the kernel will block only if the futex word has the value that the calling thread supplied (as one of the arguments of the **futex**() call) as the expected value of the futex word. The loading of the futex word's value, the comparison of that value with the expected value, and the actual blocking will happen atomically and will be totally ordered with respect to concurrent operations performed by other threads on the same futex word. Thus, the futex word is used to connect the synchronization in user space with the implementation of blocking by the kernel. Analogously to an atomic compare-and-exchange operation that potentially changes shared memory, blocking via a futex is an atomic compare-and-block operation.

One use of futexes is for implementing locks. The state of the lock (i.e., acquired or not acquired) can be represented as an atomically accessed flag in shared memory. In the uncontended case, a thread can access or modify the lock state with atomic instructions, for example atomically changing it from not acquired to acquired using an atomic compare-and-exchange instruction. (Such instructions are performed entirely in user mode, and the kernel maintains no information about the lock state.) On the other hand, a thread may be unable to acquire a lock because it is already acquired by another thread. It then may pass the lock's flag as a futex word and the value representing the acquired state as the expected value to a **futex**() wait operation. This **futex**() operation will block if and only if the lock is still acquired (i.e., the value in the futex word still matches the "acquired state"). When releasing the lock, a thread has to first reset the lock state to not acquired and then execute a futex operation that wakes threads blocked on the lock flag used as a futex word (this can be further optimized to avoid unnecessary wake-ups). See **futex**(7) for more detail on how to use futexes.

Besides the basic wait and wake-up futex functionality, there are further futex operations aimed at supporting more complex use cases.

Note that no explicit initialization or destruction is necessary to use futexes; the kernel maintains a futex (i.e., the kernel-internal implementation artifact) only while operations such as **FUTEX_WAIT**, described below, are being performed on a particular futex word.

Arguments

The *uaddr* argument points to the futex word. On all platforms, futexes are four-byte integers that must be aligned on a four-byte boundary. The operation to perform on the futex is specified in the *futex_op* argument; *val* is a value whose meaning and purpose depends on *futex_op*.

The remaining arguments (*timeout*, *uaddr2*, and *val3*) are required only for certain of the futex operations described below. Where one of these arguments is not required, it is ignored.

For several blocking operations, the *timeout* argument is a pointer to a *timespec* structure that specifies a timeout for the operation. However, notwithstanding the prototype shown above, for some operations, the least significant four bytes of this argument are instead used as an integer whose meaning is determined by the operation. For these operations, the kernel casts the *timeout* value first to *unsigned long*, then to *uint32_t*, and in the remainder of this page, this argument is referred to as *val2* when interpreted in this fashion.

Where it is required, the *uaddr2* argument is a pointer to a second futex word that is employed by the operation.

The interpretation of the final integer argument, *val3*, depends on the operation.

Futex operations

The *futex_op* argument consists of two parts: a command that specifies the operation to be performed, bitwise ORed with zero or more options that modify the behaviour of the operation. The options that may be included in *futex_op* are as follows:

FUTEX_PRIVATE_FLAG (since Linux 2.6.22)

This option bit can be employed with all futex operations. It tells the kernel that the futex is process-private and not shared with another process (i.e., it is being used for synchronization only between threads of the same process). This allows the kernel to make some additional performance optimizations.

As a convenience, *<linux/futex.h>* defines a set of constants with the suffix **_PRIVATE** that are equivalents of all of the operations listed below, but with the **FUTEX_PRIVATE_FLAG** ORed into the constant value. Thus, there are **FUTEX_WAIT_PRIVATE**, **FUTEX_WAKE_PRIVATE**, and so on.

FUTEX_CLOCK_REALTIME (since Linux 2.6.28)

This option bit can be employed only with the **FUTEX_WAIT_BITSET**, **FUTEX_WAIT_REQUEUE_PI**, and (since Linux 4.5) **FUTEX_WAIT** operations.

If this option is set, the kernel measures the *timeout* against the **CLOCK_REALTIME** clock.

If this option is not set, the kernel measures the *timeout* against the **CLOCK_MONOTONIC** clock.

The operation specified in *futex_op* is one of the following:

FUTEX_WAIT (since Linux 2.6.0)

This operation tests that the value at the futex word pointed to by the address *uaddr* still contains the expected value *val*, and if so, then sleeps waiting for a **FUTEX_WAKE** operation on the futex word. The load of the value of the futex word is an atomic memory access (i.e., using atomic machine instructions of the respective architecture). This load, the comparison with the expected value, and starting to sleep are performed atomically and totally ordered with respect to other futex operations on the same futex word. If the thread starts to sleep, it is considered a waiter on this futex word. If the futex value does not match *val*, then the call fails immediately with the error **EAGAIN**.

The purpose of the comparison with the expected value is to prevent lost wake-ups. If another thread changed the value of the futex word after the calling thread decided to block based on the prior value, and if the other thread executed a **FUTEX_WAKE** operation (or similar wake-up) after the value change and before this **FUTEX_WAIT** operation, then the calling thread will observe the value change and will not start to sleep.

If the *timeout* is not NULL, the structure it points to specifies a timeout for the wait. (This interval will be rounded up to the system clock granularity, and is guaranteed not to expire early.) The timeout is by default measured according to the **CLOCK_MONOTONIC** clock, but, since Linux 4.5, the **CLOCK_REALTIME** clock can be selected by specifying **FUTEX_CLOCK_REALTIME** in *futex_op*. If *timeout* is NULL, the call blocks indefinitely.

Note: for **FUTEX_WAIT**, *timeout* is interpreted as a *relative* value. This differs from other futex operations, where *timeout* is interpreted as an absolute value. To obtain the equivalent of **FUTEX_WAIT** with an absolute timeout, employ **FUTEX_WAIT_BITSET** with *val3* specified as **FUTEX_BITSET_MATCH_ANY**.

The arguments *uaddr2* and *val3* are ignored.

FUTEX_WAKE (since Linux 2.6.0)

This operation wakes at most *val* of the waiters that are waiting (e.g., inside **FUTEX_WAIT**) on the futex word at the address *uaddr*. Most commonly, *val* is specified as either 1 (wake up a single waiter) or **INT_MAX** (wake up all waiters). No guarantee is provided about which waiters are awoken (e.g., a waiter with a higher scheduling priority is not guaranteed to be awoken in preference to a waiter with a lower priority).

The arguments *timeout*, *uaddr2*, and *val3* are ignored.

FUTEX_FD (from Linux 2.6.0 up to and including Linux 2.6.25)

This operation creates a file descriptor that is associated with the futex at *uaddr*. The caller must close the returned file descriptor after use. When another process or thread performs a **FUTEX_WAKE** on the futex word, the file descriptor indicates as being readable with **select**(2), **poll**(2), and **epoll**(7)

The file descriptor can be used to obtain asynchronous notifications: if *val* is nonzero, then, when another process or thread executes a **FUTEX_WAKE**, the caller will receive the signal number that was passed in *val*.

The arguments *timeout*, *uaddr2* and *val3* are ignored.

Because it was inherently racy, **FUTEX_FD** has been removed from Linux 2.6.26 onward.

FUTEX_REQUEUE (since Linux 2.6.0)

This operation performs the same task as **FUTEX_CMP_REQUEUE** (see below), except that no check is made using the value in *val3*. (The argument *val3* is ignored.)

FUTEX_CMP_REQUEUE (since Linux 2.6.7)

This operation first checks whether the location *uaddr* still contains the value *val3*. If not, the operation fails with the error **EAGAIN**. Otherwise, the operation wakes up a maximum of *val* waiters that are waiting on the futex at *uaddr*. If there are more than *val* waiters, then the remaining waiters are removed from the wait queue of the source futex at *uaddr* and added to the wait queue of the target futex at *uaddr2*. The *val2* argument specifies an upper limit on the number of waiters that are requeued to the futex at *uaddr2*.

The load from *uaddr* is an atomic memory access (i.e., using atomic machine instructions of the respective architecture). This load, the comparison with *val3*, and the requeueing of any waiters are performed atomically and totally ordered with respect to other operations on the same futex word.

Typical values to specify for *val* are 0 or 1. (Specifying **INT_MAX** is not useful, because it would make the **FUTEX_CMP_REQUEUE** operation equivalent to **FUTEX_WAKE**.) The limit value specified via *val2* is typically either 1 or **INT_MAX**. (Specifying the argument as 0 is not useful, because it would make the **FUTEX_CMP_REQUEUE** operation equivalent to **FUTEX_WAIT**.)

The **FUTEX_CMP_REQUEUE** operation was added as a replacement for the earlier **FUTEX_REQUEUE**. The difference is that the check of the value at *uaddr* can be used to ensure that requeueing happens only under certain conditions, which allows race conditions to be avoided in certain use cases.

Both **FUTEX_REQUEUE** and **FUTEX_CMP_REQUEUE** can be used to avoid "thundering herd" wake-ups that could occur when using **FUTEX_WAKE** in cases where all of the waiters that are woken need to acquire another futex. Consider the following scenario, where multiple waiter threads are waiting on B, a wait queue implemented using a futex:

```
lock(A)
while (!check_value(V)) {
    unlock(A);
    block_on(B);
    lock(A);
};
unlock(A);
```

If a waker thread used **FUTEX_WAKE**, then all waiters waiting on B would be woken up, and they would all try to acquire lock A. However, waking all of the threads in this manner would be pointless because all except one of the threads would immediately block on lock A again. By contrast, a requeue operation wakes just one waiter and moves the other waiters to lock A, and when the woken waiter unlocks A then the next waiter can proceed.

FUTEX_WAKE_OP (since Linux 2.6.14)

This operation was added to support some user-space use cases where more than one futex must be handled at the same time. The most notable example is the implementation of **pthread_cond_signal**(3), which requires operations on two futexes, the one used to implement the mutex and the one used in the implementation of the wait queue associated with the condition variable. **FUTEX_WAKE_OP** allows such cases to be implemented without leading to high rates of contention and context switching.

The **FUTEX_WAKE_OP** operation is equivalent to executing the following code atomically and totally ordered with respect to other futex operations on any of the two supplied futex words:

```
int oldval = *(int *) uaddr2;
*(int *) uaddr2 = oldval op oparg;
futex(uaddr, FUTEX_WAKE, val, 0, 0, 0);
if (oldval cmp cmparg)
    futex(uaddr2, FUTEX_WAKE, val2, 0, 0, 0);
```

In other words, **FUTEX_WAKE_OP** does the following:

* saves the original value of the futex word at *uaddr2* and performs an operation to modify the value of the futex at *uaddr2*; this is an atomic read-modify-write memory access (i.e., using atomic machine instructions of the respective architecture)

* wakes up a maximum of *val* waiters on the futex for the futex word at *uaddr*; and

* dependent on the results of a test of the original value of the futex word at *uaddr2*, wakes up a maximum of *val2* waiters on the futex for the futex word at *uaddr2*.

The operation and comparison that are to be performed are encoded in the bits of the argument *val3*. Pictorially, the encoding is:

```
    +---+---+-----------+-----------+
    |op |cmp|   oparg   |  cmparg   |
    +---+---+-----------+-----------+
      4   4       12          12    <== # of bits
```

Expressed in code, the encoding is:

```
#define FUTEX_OP(op, oparg, cmp, cmparg) \
                (((op & 0xf) << 28) | \
                ((cmp & 0xf) << 24) | \
                ((oparg & 0xfff) << 12) | \
                      (cmparg & 0xfff))
```

In the above, *op* and *cmp* are each one of the codes listed below. The *oparg* and *cmparg* components are literal numeric values, except as noted below.

The *op* component has one of the following values:

```
FUTEX_OP_SET        0  /* uaddr2 = oparg; */
FUTEX_OP_ADD        1  /* uaddr2 += oparg; */
FUTEX_OP_OR         2  /* uaddr2 |= oparg; */
FUTEX_OP_ANDN       3  /* uaddr2 &= ~oparg; */
FUTEX_OP_XOR        4  /* uaddr2 ^= oparg; */
```

In addition, bit-wise ORing the following value into *op* causes *(1 << oparg)* to be used as the operand:

```
FUTEX_OP_ARG_SHIFT  8  /* Use (1 << oparg) as operand */
```

The *cmp* field is one of the following:

```
FUTEX_OP_CMP_EQ     0  /* if (oldval == cmparg) wake */
FUTEX_OP_CMP_NE     1  /* if (oldval != cmparg) wake */
FUTEX_OP_CMP_LT     2  /* if (oldval < cmparg) wake */
FUTEX_OP_CMP_LE     3  /* if (oldval <= cmparg) wake */
FUTEX_OP_CMP_GT     4  /* if (oldval > cmparg) wake */
FUTEX_OP_CMP_GE     5  /* if (oldval >= cmparg) wake */
```

The return value of **FUTEX_WAKE_OP** is the sum of the number of waiters woken on the futex *uaddr* plus the number of waiters woken on the futex *uaddr2*.

FUTEX_WAIT_BITSET (since Linux 2.6.25)

This operation is like **FUTEX_WAIT** except that *val3* is used to provide a 32-bit bit mask to the kernel. This bit mask, in which at least one bit must be set, is stored in the kernel-internal state of the waiter. See the description of **FUTEX_WAKE_BITSET** for further details.

If *timeout* is not NULL, the structure it points to specifies an absolute timeout for the wait operation. If *timeout* is NULL, the operation can block indefinitely.

The *uaddr2* argument is ignored.

FUTEX_WAKE_BITSET (since Linux 2.6.25)

This operation is the same as **FUTEX_WAKE** except that the *val3* argument is used to provide a 32-bit bit mask to the kernel. This bit mask, in which at least one bit must be set, is used to select which waiters should be woken up. The selection is done by a bit-wise AND of the "wake" bit mask (i.e., the value in *val3*) and the bit mask which is stored in the kernel-internal state of the waiter (the "wait" bit mask that is set using **FUTEX_WAIT_BITSET**). All of the waiters for which the result of the AND is nonzero are woken up; the remaining waiters are left sleeping.

The effect of **FUTEX_WAIT_BITSET** and **FUTEX_WAKE_BITSET** is to allow selective wake-ups among multiple waiters that are blocked on the same futex. However, note that, depending on the use case, employing this bit-mask multiplexing feature on a futex can be less efficient than simply using multiple futexes, because employing bit-mask multiplexing requires the kernel to check all waiters on a futex, including those that are not interested in being woken up (i.e., they do not have the relevant bit set in their "wait" bit mask).

The constant **FUTEX_BITSET_MATCH_ANY**, which corresponds to all 32 bits set in the bit mask, can be used as the *val3* argument for **FUTEX_WAIT_BITSET** and **FUTEX_WAKE_BITSET**. Other than differences in the handling of the *timeout* argument, the **FUTEX_WAIT** operation is equivalent to **FUTEX_WAIT_BITSET** with *val3* specified as **FUTEX_BITSET_MATCH_ANY**; that is, allow a wake-up by any waker. The **FUTEX_WAKE** operation is equivalent to **FUTEX_WAKE_BITSET** with *val3* specified as **FUTEX_BITSET_MATCH_ANY**; that is, wake up any waiter(s).

The *uaddr2* and *timeout* arguments are ignored.

Priority-inheritance futexes

Linux supports priority-inheritance (PI) futexes in order to handle priority-inversion problems that can be encountered with normal futex locks. Priority inversion is the problem that occurs when a high-priority task is blocked waiting to acquire a lock held by a low-priority task, while tasks at an intermediate priority continuously preempt the low-priority task from the CPU. Consequently, the low-priority task makes no progress toward releasing the lock, and the high-priority task remains blocked.

Priority inheritance is a mechanism for dealing with the priority-inversion problem. With this mechanism, when a high-priority task becomes blocked by a lock held by a low-priority task, the priority of the low-priority task is temporarily raised to that of the high-priority task, so that it is not preempted by any intermediate level tasks, and can thus make progress toward releasing the lock. To be effective, priority inheritance must be transitive, meaning that if a high-priority task blocks on a lock held by a lower-priority task that is itself blocked by a lock held by another intermediate-priority task (and so on, for chains of arbitrary length), then both of those tasks (or more generally, all of the tasks in a lock chain) have their priorities raised to be the same as the high-priority task.

From a user-space perspective, what makes a futex PI-aware is a policy agreement (described below) between user space and the kernel about the value of the futex word, coupled with the use of the PI-futex operations described below. (Unlike the other futex operations described above, the PI-futex operations are designed for the implementation of very specific IPC mechanisms.)

The PI-futex operations described below differ from the other futex operations in that they impose policy on the use of the value of the futex word:

* If the lock is not acquired, the futex word's value shall be 0.
* If the lock is acquired, the futex word's value shall be the thread ID (TID; see **gettid**(2)) of the owning thread.
* If the lock is owned and there are threads contending for the lock, then the **FUTEX_WAITERS** bit shall be set in the futex word's value; in other words, this value is:

  FUTEX_WAITERS | TID

  (Note that is invalid for a PI futex word to have no owner and **FUTEX_WAITERS** set.)

With this policy in place, a user-space application can acquire an unacquired lock or release a lock using atomic instructions executed in user mode (e.g., a compare-and-swap operation such as *cmpxchg* on the x86 architecture). Acquiring a lock simply consists of using compare-and-swap to atomically set the futex word's value to the caller's TID if its previous value was 0. Releasing a lock requires using compare-and-swap to set the futex word's value to 0 if the previous value was the expected TID.

If a futex is already acquired (i.e., has a nonzero value), waiters must employ the **FUTEX_LOCK_PI** operation to acquire the lock. If other threads are waiting for the lock, then the **FUTEX_WAITERS** bit is set in the futex value; in this case, the lock owner must employ the **FUTEX_UNLOCK_PI** operation to release the lock.

In the cases where callers are forced into the kernel (i.e., required to perform a **futex**() call), they then deal directly with a so-called RT-mutex, a kernel locking mechanism which implements the required priority-inheritance semantics. After the RT-mutex is acquired, the futex value is updated accordingly, before the calling thread returns to user space.

It is important to note that the kernel will update the futex word's value prior to returning to user space. (This prevents the possibility of the futex word's value ending up in an invalid state, such as having an owner but the value being 0, or having waiters but not having the **FUTEX_WAITERS** bit set.)

If a futex has an associated RT-mutex in the kernel (i.e., there are blocked waiters) and the owner of the futex/RT-mutex dies unexpectedly, then the kernel cleans up the RT-mutex and hands it over to the next waiter. This in turn requires that the user-space value is updated accordingly. To indicate that this is required, the kernel sets the **FUTEX_OWNER_DIED** bit in the futex word along with the thread ID of the new owner. User space can detect this situation via the presence of the **FUTEX_OWNER_DIED** bit and is then responsible for cleaning up the stale state left over by the dead owner.

PI futexes are operated on by specifying one of the values listed below in *futex_op*. Note that the PI futex operations must be used as paired operations and are subject to some additional requirements:

* **FUTEX_LOCK_PI** and **FUTEX_TRYLOCK_PI** pair with **FUTEX_UNLOCK_PI**. **FUTEX_UNLOCK_PI** must be called only on a futex owned by the calling thread, as defined by the value policy, otherwise the error **EPERM** results.
* **FUTEX_WAIT_REQUEUE_PI** pairs with **FUTEX_CMP_REQUEUE_PI**. This must be performed from a non-PI futex to a distinct PI futex (or the error **EINVAL** results). Additionally, *val* (the number of waiters to be woken) must be 1 (or the error **EINVAL** results).

The PI futex operations are as follows:

FUTEX_LOCK_PI (since Linux 2.6.18)

This operation is used after an attempt to acquire the lock via an atomic user-mode instruction failed because the futex word has a nonzero value—specifically, because it contained the (PID-namespace-specific) TID of the lock owner.

The operation checks the value of the futex word at the address *uaddr*. If the value is 0, then the kernel tries to atomically set the futex value to the caller's TID. If the futex word's value is nonzero, the kernel atomically sets the **FUTEX_WAITERS** bit, which signals the futex owner that it cannot unlock the futex in user space atomically by setting the futex value to 0. After that, the kernel:

1. Tries to find the thread which is associated with the owner TID.
2. Creates or reuses kernel state on behalf of the owner. (If this is the first waiter, there is no kernel state for this futex, so kernel state is created by locking the RT-mutex and the futex owner is made the owner of the RT-mutex. If there are existing waiters, then the existing state is reused.)
3. Attaches the waiter to the futex (i.e., the waiter is enqueued on the RT-mutex waiter list).

If more than one waiter exists, the enqueueing of the waiter is in descending priority order. (For information on priority ordering, see the discussion of the **SCHED_DEADLINE**, **SCHED_FIFO**, and **SCHED_RR** scheduling policies in **sched**(7).) The owner inherits either the waiter's CPU bandwidth (if the waiter is scheduled under the **SCHED_DEADLINE** policy) or the waiter's priority (if the waiter is scheduled under the **SCHED_RR** or **SCHED_FIFO** policy). This inheritance follows the lock chain in the case of nested locking and performs deadlock detection.

The *timeout* argument provides a timeout for the lock attempt. If *timeout* is not NULL, the structure it points to specifies an absolute timeout, measured against the **CLOCK_REALTIME** clock. If *timeout* is NULL, the operation will block indefinitely.

The *uaddr2*, *val*, and *val3* arguments are ignored.

FUTEX_TRYLOCK_PI (since Linux 2.6.18)

This operation tries to acquire the lock at *uaddr*. It is invoked when a user-space atomic acquire did not succeed because the futex word was not 0.

Because the kernel has access to more state information than user space, acquisition of the lock might succeed if performed by the kernel in cases where the futex word (i.e., the state information accessible to use-space) contains stale state (**FUTEX_WAITERS** and/or **FUTEX_OWNER_DIED**). This can happen when the owner of the futex died. User space cannot handle this condition in a race-free manner, but the kernel can fix this up and acquire the futex.

The *uaddr2*, *val*, *timeout*, and *val3* arguments are ignored.

FUTEX_UNLOCK_PI (since Linux 2.6.18)

This operation wakes the top priority waiter that is waiting in **FUTEX_LOCK_PI** on the futex address provided by the *uaddr* argument.

This is called when the user-space value at *uaddr* cannot be changed atomically from a TID (of the owner) to 0.

The *uaddr2*, *val*, *timeout*, and *val3* arguments are ignored.

FUTEX_CMP_REQUEUE_PI (since Linux 2.6.31)

This operation is a PI-aware variant of **FUTEX_CMP_REQUEUE**. It requeues waiters that are blocked via **FUTEX_WAIT_REQUEUE_PI** on *uaddr* from a non-PI source futex (*uaddr*) to a PI target futex (*uaddr2*).

As with **FUTEX_CMP_REQUEUE**, this operation wakes up a maximum of *val* waiters that are waiting on the futex at *uaddr*. However, for **FUTEX_CMP_REQUEUE_PI**, *val* is required to be 1 (since the main point is to avoid a thundering herd). The remaining waiters are removed from the wait queue of the source futex at *uaddr* and added to the wait queue of the target futex at *uaddr2*.

The *val2* and *val3* arguments serve the same purposes as for **FUTEX_CMP_REQUEUE**.

FUTEX_WAIT_REQUEUE_PI (since Linux 2.6.31)

Wait on a non-PI futex at *uaddr* and potentially be requeued (via a **FUTEX_CMP_REQUEUE_PI** operation in another task) onto a PI futex at *uaddr2*. The wait operation on *uaddr* is the same as for **FUTEX_WAIT**.

The waiter can be removed from the wait on *uaddr* without requeueing on *uaddr2* via a **FUTEX_WAKE** operation in another task. In this case, the **FUTEX_WAIT_REQUEUE_PI** operation fails with the error **EAGAIN**.

If *timeout* is not NULL, the structure it points to specifies an absolute timeout for the wait operation. If *timeout* is NULL, the operation can block indefinitely.

The *val3* argument is ignored.

The **FUTEX_WAIT_REQUEUE_PI** and **FUTEX_CMP_REQUEUE_PI** were added to support a fairly specific use case: support for priority-inheritance-aware POSIX threads condition variables. The idea is that these operations should always be paired, in order to ensure that user space and the kernel remain in sync. Thus, in the **FUTEX_WAIT_REQUEUE_PI** operation, the user-space application pre-specifies the target of the requeue that takes place in the **FUTEX_CMP_REQUEUE_PI** operation.

RETURN VALUE

In the event of an error (and assuming that **futex**() was invoked via **syscall**(2)), all operations return −1 and set *errno* to indicate the cause of the error.

The return value on success depends on the operation, as described in the following list:

FUTEX_WAIT

Returns 0 if the caller was woken up. Note that a wake-up can also be caused by common futex usage patterns in unrelated code that happened to have previously used the futex word's memory location (e.g., typical futex-based implementations of Pthreads mutexes can cause this under some conditions). Therefore, callers should always conservatively assume that a return value of 0 can mean a spurious wake-up, and use the futex word's value (i.e., the user-space synchronization scheme) to decide whether to continue to block or not.

FUTEX_WAKE

Returns the number of waiters that were woken up.

FUTEX_FD

Returns the new file descriptor associated with the futex.

FUTEX_REQUEUE

Returns the number of waiters that were woken up.

FUTEX_CMP_REQUEUE
Returns the total number of waiters that were woken up or requeued to the futex for the futex word at *uaddr2*. If this value is greater than *val*, then the difference is the number of waiters requeued to the futex for the futex word at *uaddr2*.

FUTEX_WAKE_OP
Returns the total number of waiters that were woken up. This is the sum of the woken waiters on the two futexes for the futex words at *uaddr* and *uaddr2*.

FUTEX_WAIT_BITSET
Returns 0 if the caller was woken up. See **FUTEX_WAIT** for how to interpret this correctly in practice.

FUTEX_WAKE_BITSET
Returns the number of waiters that were woken up.

FUTEX_LOCK_PI
Returns 0 if the futex was successfully locked.

FUTEX_TRYLOCK_PI
Returns 0 if the futex was successfully locked.

FUTEX_UNLOCK_PI
Returns 0 if the futex was successfully unlocked.

FUTEX_CMP_REQUEUE_PI
Returns the total number of waiters that were woken up or requeued to the futex for the futex word at *uaddr2*. If this value is greater than *val*, then difference is the number of waiters requeued to the futex for the futex word at *uaddr2*.

FUTEX_WAIT_REQUEUE_PI
Returns 0 if the caller was successfully requeued to the futex for the futex word at *uaddr2*.

ERRORS

EACCES
No read access to the memory of a futex word.

EAGAIN
(**FUTEX_WAIT**, **FUTEX_WAIT_BITSET**, **FUTEX_WAIT_REQUEUE_PI**) The value pointed to by *uaddr* was not equal to the expected value *val* at the time of the call.

Note: on Linux, the symbolic names **EAGAIN** and **EWOULDBLOCK** (both of which appear in different parts of the kernel futex code) have the same value.

EAGAIN
(**FUTEX_CMP_REQUEUE**, **FUTEX_CMP_REQUEUE_PI**) The value pointed to by *uaddr* is not equal to the expected value *val3*.

EAGAIN
(**FUTEX_LOCK_PI**, **FUTEX_TRYLOCK_PI**, **FUTEX_CMP_REQUEUE_PI**) The futex owner thread ID of *uaddr* (for **FUTEX_CMP_REQUEUE_PI**: *uaddr2*) is about to exit, but has not yet handled the internal state cleanup. Try again.

EDEADLK
(**FUTEX_LOCK_PI**, **FUTEX_TRYLOCK_PI**, **FUTEX_CMP_REQUEUE_PI**) The futex word at *uaddr* is already locked by the caller.

EDEADLK
(**FUTEX_CMP_REQUEUE_PI**) While requeueing a waiter to the PI futex for the futex word at *uaddr2*, the kernel detected a deadlock.

EFAULT
A required pointer argument (i.e., *uaddr*, *uaddr2*, or *timeout*) did not point to a valid user-space address.

EINTR
A **FUTEX_WAIT** or **FUTEX_WAIT_BITSET** operation was interrupted by a signal (see **signal**(7)). In kernels before Linux 2.6.22, this error could also be returned for a spurious wakeup; since Linux 2.6.22, this no longer happens.

EINVAL
The operation in *futex_op* is one of those that employs a timeout, but the supplied *timeout* argument was invalid (*tv_sec* was less than zero, or *tv_nsec* was not less than 1,000,000,000).

EINVAL
The operation specified in *futex_op* employs one or both of the pointers *uaddr* and *uaddr2*, but one of these does not point to a valid object—that is, the address is not four-byte-aligned.

EINVAL
(**FUTEX_WAIT_BITSET**, **FUTEX_WAKE_BITSET**) The bit mask supplied in *val3* is zero.

EINVAL
(**FUTEX_CMP_REQUEUE_PI**) *uaddr* equals *uaddr2* (i.e., an attempt was made to requeue to the same futex).

EINVAL
(**FUTEX_FD**) The signal number supplied in *val* is invalid.

EINVAL
(**FUTEX_WAKE**, **FUTEX_WAKE_OP**, **FUTEX_WAKE_BITSET**, **FUTEX_REQUEUE**, **FUTEX_CMP_REQUEUE**) The kernel detected an inconsistency between the user-space state at *uaddr* and the kernel state—that is, it detected a waiter which waits in **FUTEX_LOCK_PI** on *uaddr*.

EINVAL
(**FUTEX_LOCK_PI**, **FUTEX_TRYLOCK_PI**, **FUTEX_UNLOCK_PI**) The kernel detected an inconsistency between the user-space state at *uaddr* and the kernel state. This indicates either state corruption or that the kernel found a waiter on *uaddr* which is waiting via **FUTEX_WAIT** or **FUTEX_WAIT_BITSET**.

EINVAL
(**FUTEX_CMP_REQUEUE_PI**) The kernel detected an inconsistency between the user-space state at *uaddr2* and the kernel state; that is, the kernel detected a waiter which waits via **FUTEX_WAIT** or **FUTEX_WAIT_BITSET** on *uaddr2*.

EINVAL
(**FUTEX_CMP_REQUEUE_PI**) The kernel detected an inconsistency between the user-space state at *uaddr* and the kernel state; that is, the kernel detected a waiter which waits via **FUTEX_WAIT** or **FUTEX_WAIT_BITESET** on *uaddr*.

EINVAL
(**FUTEX_CMP_REQUEUE_PI**) The kernel detected an inconsistency between the user-space state at *uaddr* and the kernel state; that is, the kernel detected a waiter which waits on *uaddr* via **FUTEX_LOCK_PI** (instead of **FUTEX_WAIT_REQUEUE_PI**).

EINVAL
(**FUTEX_CMP_REQUEUE_PI**) An attempt was made to requeue a waiter to a futex other than that specified by the matching **FUTEX_WAIT_REQUEUE_PI** call for that waiter.

EINVAL
(**FUTEX_CMP_REQUEUE_PI**) The *val* argument is not 1.

EINVAL
Invalid argument.

ENFILE
(**FUTEX_FD**) The system-wide limit on the total number of open files has been reached.

ENOMEM
(**FUTEX_LOCK_PI**, **FUTEX_TRYLOCK_PI**, **FUTEX_CMP_REQUEUE_PI**) The kernel could not allocate memory to hold state information.

ENOSYS
Invalid operation specified in *futex_op*.

ENOSYS
The **FUTEX_CLOCK_REALTIME** option was specified in *futex_op*, but the accompanying operation was neither **FUTEX_WAIT**, **FUTEX_WAIT_BITSET**, nor **FUTEX_WAIT_REQUEUE_PI**.

ENOSYS
(**FUTEX_LOCK_PI**, **FUTEX_TRYLOCK_PI**, **FUTEX_UNLOCK_PI**, **FUTEX_CMP_REQUEUE_PI**, **FUTEX_WAIT_REQUEUE_PI**) A run-time check determined that the operation is not available. The PI-futex operations are not implemented on all architectures and are not supported on some CPU variants.

EPERM
(**FUTEX_LOCK_PI**, **FUTEX_TRYLOCK_PI**, **FUTEX_CMP_REQUEUE_PI**) The caller is not allowed to attach itself to the futex at *uaddr* (for **FUTEX_CMP_REQUEUE_PI**: the futex at *uaddr2*). (This may be caused by a state corruption in user space.)

EPERM
(**FUTEX_UNLOCK_PI**) The caller does not own the lock represented by the futex word.

ESRCH
(**FUTEX_LOCK_PI**, **FUTEX_TRYLOCK_PI**, **FUTEX_CMP_REQUEUE_PI**) The thread ID in the futex word at *uaddr* does not exist.

ESRCH
(**FUTEX_CMP_REQUEUE_PI**) The thread ID in the futex word at *uaddr2* does not exist.

ETIMEDOUT
The operation in *futex_op* employed the timeout specified in *timeout*, and the timeout expired before the operation completed.

VERSIONS

Futexes were first made available in a stable kernel release with Linux 2.6.0.

Initial futex support was merged in Linux 2.5.7 but with different semantics from what was described above. A four-argument system call with the semantics described in this page was introduced in Linux 2.5.40. A fifth argument was added in Linux 2.5.70, and a sixth argument was added in Linux 2.6.7.

CONFORMING TO

This system call is Linux-specific.

NOTES

Glibc does not provide a wrapper for this system call; call it using **syscall**(2).

Several higher-level programming abstractions are implemented via futexes, including POSIX semaphores and various POSIX threads synchronization mechanisms (mutexes, condition variables, read-write locks, and barriers).

EXAMPLE

The program below demonstrates use of futexes in a program where a parent process and a child process use a pair of futexes located inside a shared anonymous mapping to synchronize access to a shared resource: the terminal. The two processes each write *nloops* (a command-line argument that defaults to 5 if omitted) messages to the terminal and employ a synchronization protocol that ensures that they alternate in writing messages. Upon running this program we see output such as the following:

```
$ ./futex_demo
Parent (18534) 0
```

```
    Child  (18535) 0
    Parent (18534) 1
    Child  (18535) 1
    Parent (18534) 2
    Child  (18535) 2
    Parent (18534) 3
    Child  (18535) 3
    Parent (18534) 4
    Child  (18535) 4
```

Program source

```
/* futex_demo.c

   Usage: futex_demo [nloops]
                    (Default: 5)

   Demonstrate the use of futexes in a program where parent and child
   use a pair of futexes located inside a shared anonymous mapping to
   synchronize access to a shared resource: the terminal. The two
   processes each write 'num-loops' messages to the terminal and employ
   a synchronization protocol that ensures that they alternate in
   writing messages.
*/
#define _GNU_SOURCE
#include <stdio.h>
#include <errno.h>
#include <stdatomic.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>
#include <sys/mman.h>
#include <sys/syscall.h>
#include <linux/futex.h>
#include <sys/time.h>

#define errExit(msg)    do { perror(msg); exit(EXIT_FAILURE); \
                        } while (0)

static int *futex1, *futex2, *iaddr;

static int
futex(int *uaddr, int futex_op, int val,
      const struct timespec *timeout, int *uaddr2, int val3)
{
    return syscall(SYS_futex, uaddr, futex_op, val,
                   timeout, uaddr, val3);
}

/* Acquire the futex pointed to by 'futexp': wait for its value to
   become 1, and then set the value to 0. */

static void
fwait(int *futexp)
{
    int s;

    /* atomic_compare_exchange_strong(ptr, oldval, newval)
       atomically performs the equivalent of:

           if (*ptr == *oldval)
               *ptr = newval;

       It returns true if the test yielded true and *ptr was updated. */

    while (1) {

        /* Is the futex available? */
        const int zero = 0;
        if (atomic_compare_exchange_strong(futexp, &zero, 1))
            break;      /* Yes */

        /* Futex is not available; wait */

        s = futex(futexp, FUTEX_WAIT, 0, NULL, NULL, 0);
        if (s == -1 && errno != EAGAIN)
            errExit("futex-FUTEX_WAIT");
    }
}

/* Release the futex pointed to by 'futexp': if the futex currently
   has the value 0, set its value to 1 and the wake any futex waiters,
   so that if the peer is blocked in fpost(), it can proceed. */

static void
fpost(int *futexp)
{
    int s;

    /* atomic_compare_exchange_strong() was described in comments above */

    const int one = 1;
    if (atomic_compare_exchange_strong(futexp, &one, 0)) {
        s = futex(futexp, FUTEX_WAKE, 1, NULL, NULL, 0);
        if (s  == -1)
            errExit("futex-FUTEX_WAKE");
    }
}

int
main(int argc, char *argv[])
{
    pid_t childPid;
    int j, nloops;

    setbuf(stdout, NULL);

    nloops = (argc > 1) ? atoi(argv[1]) : 5;
    /* Create a shared anonymous mapping that will hold the futexes.
       Since the futexes are being shared between processes, we
       subsequently use the "shared" futex operations (i.e., not the
       ones suffixed "_PRIVATE") */

    iaddr = mmap(NULL, sizeof(int) * 2, PROT_READ | PROT_WRITE,
                MAP_ANONYMOUS | MAP_SHARED, -1, 0);
    if (iaddr == MAP_FAILED)
        errExit("mmap");

    futex1 = &iaddr[0];
    futex2 = &iaddr[1];

    *futex1 = 0;        /* State: unavailable */
    *futex2 = 1;        /* State: available */

    /* Create a child process that inherits the shared anonymous
       mapping */

    childPid = fork();
    if (childPid == -1)
        errExit("fork");

    if (childPid == 0) {        /* Child */
        for (j = 0; j < nloops; j++) {
            fwait(futex1);
            printf("Child  (%ld) %d\n", (long) getpid(), j);
            fpost(futex2);
        }

        exit(EXIT_SUCCESS);
    }

    /* Parent falls through to here */

    for (j = 0; j < nloops; j++) {
        fwait(futex2);
        printf("Parent (%ld) %d\n", (long) getpid(), j);
        fpost(futex1);
    }

    wait(NULL);

    exit(EXIT_SUCCESS);
}
```

SEE ALSO

get_robust_list(2), **restart_syscall**(2), **pthread_mutexattr_getprotocol**(3), **futex**(7), **sched**(7)

The following kernel source files:

* *Documentation/pi-futex.txt*

* *Documentation/futex-requeue-pi.txt*

* *Documentation/locking/rt-mutex.txt*

* *Documentation/locking/rt-mutex-design.txt*

* *Documentation/robust-futex-ABI.txt*

Franke, H., Russell, R., and Kirwood, M., 2002. *Fuss, Futexes and Furwocks: Fast Userlevel Locking in Linux* (from proceedings of the Ottawa Linux Symposium 2002), ⟨http://kernel.org/doc/ols/2002/ols2002−pages−479−495.pdf⟩

Hart, D., 2009. *A futex overview and update*, ⟨http://lwn.net/Articles/360699/⟩

Hart, D. and Guniguntala, D., 2009. *Requeue-PI: Making Glibc Condvars PI-Aware* (from proceedings of the 2009 Real-Time Linux Workshop), ⟨http://lwn.net/images/conf/rtlws11/papers/proc/p10.pdf⟩

Drepper, U., 2011. *Futexes Are Tricky*, ⟨http://www.akkadia.org/drepper/futex.pdf⟩

Futex example library, futex-*.tar.bz2 at ⟨ftp://ftp.kernel.org/pub/linux/kernel/people/rusty/⟩

COLOPHON

This page is part of release 5.02 of the Linux *man-pages* project. A description of the project, information about reporting bugs, and the latest version of this page, can be found at https://www.kernel.org/doc/man−pages/.