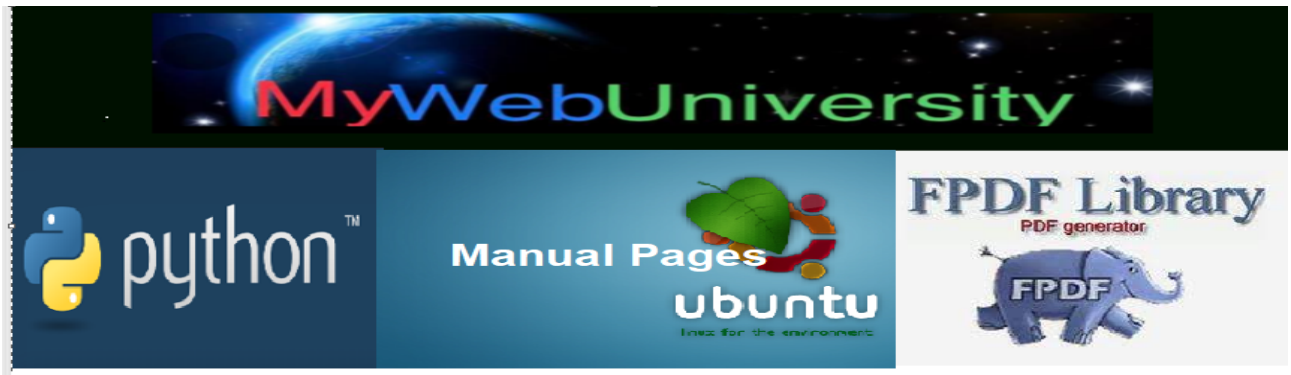

Full credit is given to the above companies including the OS that this PDF file was generated!

Linux Ubuntu 26.04 Manual Pages on command 'futex.2'

$ man futex.2

futex(2) System Calls Manual futex(2)

NAME

futex - fast user-space locking

LIBRARY

Standard C library (libc, -lc)

SYNOPSIS

```
#include <linux/futex.h>  /* Definition of FUTEX_* constants */
#include <sys/syscall.h>  /* Definition of SYS_* constants */
#include <unistd.h>
long syscall(SYS_futex, uint32_t *uaddr, int op, ...);
```

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 synchro? nization. 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 fu? tex() operations can be used to wake any processes or threads waiting for a particular con? dition.

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 sys? tems.) 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 differ?

ent 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 per? formed 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. Analo? gously 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 uncon? tended case, a thread can access or modify the lock state with atomic instructions, for ex? ample atomically changing it from not acquired to acquired using an atomic compare-and-ex? change 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 ac? quired 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(2const) are being performed on a particular futex word.

Arguments

The uaddr argument points to the futex word. On all platforms, futexes are four-byte inte?

gers that must be aligned on a four-byte boundary. The operation to perform on the futex is specified in the op argument.

Futex operations

The 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 behavior of the operation. The options that may be included in 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(2const), FUTEX_WAIT_REQUEUE_PI(2const), (since Linux 4.5) FUTEX_WAIT(2const), and (since Linux 5.14) FUTEX_LOCK_PI2(2const) 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 op is one of the following:

FUTEX_WAIT(2const)

FUTEX_WAKE(2const)

FUTEX_FD(2const)

FUTEX_REQUEUE(2const)

FUTEX_CMP_REQUEUE(2const)

FUTEX_WAKE_OP(2const)

FUTEX_WAIT_BITSET(2const)

FUTEX_WAKE_BITSET(2const)

Priority-inheritance futexes

Linux supports priority-inheritance (PI) futexes in order to handle priority-inversion prob?
lems 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-prior?
ity 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 transi?
tive, 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 de?
scribed 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 get?
tid(2)) of the owning thread.

? If the lock is owned and there are threads contending for the lock, then the FUTEX_WAIT?
ERS 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 re?
lease a lock using atomic instructions executed in user mode (e.g., a compare-and-swap oper?
ation 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 previ?

ous 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 FU? TEX_LOCK_PI(2const) or FUTEX_LOCK_PI2(2const) operations 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(2const) 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 return? ing 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 hav? ing 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 FU? TEX_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 responsi? ble 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 op. Note that the PI futex operations must be used as paired operations and are subject to some additional requirements:

? FUTEX_LOCK_PI(2const), FUTEX_LOCK_PI2(2const), and FUTEX_TRYLOCK_PI(2const) pair with FU? TEX_UNLOCK_PI(2const). FUTEX_UNLOCK_PI(2const) 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(2const) pairs with FUTEX_CMP_REQUEUE_PI(2const). This must be per? formed from a non-PI futex to a distinct PI futex (or the error EINVAL results). Addi? tionally, 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(2const)

FUTEX_LOCK_PI2(2const)

FUTEX_TRYLOCK_PI(2const)

FUTEX_UNLOCK_PI(2const)

FUTEX_CMP_REQUEUE_PI(2const)

FUTEX_WAIT_REQUEUE_PI(2const)

The FUTEX_WAIT_REQUEUE_PI(2const) and FUTEX_CMP_REQUEUE_PI(2const) 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(2const) operation, the user-space application pre-specifies the target of the requeue that takes place in the FUTEX_CMP_REQUEUE_PI(2const) operation.

RETURN VALUE

On error, -1 is returned, and errno is set to indicate the error.

The return value on success depends on the operation.

ERRORS

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

EFAULT uaddr did not point to a valid user-space address.

EINVAL uaddr does not point to a valid object?that is, the address is not four-byte-aligned.

EINVAL Invalid argument.

ENOSYS Invalid operation specified in op.

ENOSYS The FUTEX_CLOCK_REALTIME option was specified in op, but the accompanying operation was neither FUTEX_WAIT_BITSET(2const), FUTEX_WAIT_REQUEUE_PI(2const), nor FU? TEX_LOCK_PI2(2const).

STANDARDS

Linux.

HISTORY

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 argu? ment was added in Linux 2.6.7.

EXAMPLES

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 synchro? nization 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 <err.h>
#include <errno.h>
#include <linux/futex.h>
#include <stdatomic.h>
#include <stdint.h>
```

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/mman.h>
#include <sys/syscall.h>
#include <sys/time.h>
#include <sys/wait.h>
#include <unistd.h>
static uint32_t *futex1, *futex2, *iaddr;
static int
futex(uint32_t *uaddr, int op, uint32_t val,
      const struct timespec *timeout, uint32_t *uaddr2, uint32_t val3)
{
    return syscall(SYS_futex, uaddr, op, val,
                   timeout, uaddr2, 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(uint32_t *futexp)
{
    long            s;
    const uint32_t  one = 1;
    /* 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? */
        if (atomic_compare_exchange_strong(futexp, &one, 0))
            break;      /* Yes */
        /* Futex is not available; wait.  */
        s = futex(futexp, FUTEX_WAIT, 0, NULL, NULL, 0);
```

```
        if (s == -1 && errno != EAGAIN)
            err(EXIT_FAILURE, "futex-FUTEX_WAIT");
    }
}
/* Release the futex pointed to by 'futexp': if the futex currently
   has the value 0, set its value to 1 and then wake any futex waiters,
   so that if the peer is blocked in fwait(), it can proceed.  */
static void
fpost(uint32_t *futexp)
{
    long            s;
    const uint32_t  zero = 0;
    /* atomic_compare_exchange_strong() was described
       in comments above.  */
    if (atomic_compare_exchange_strong(futexp, &zero, 1)) {
        s = futex(futexp, FUTEX_WAKE, 1, NULL, NULL, 0);
        if (s  == -1)
            err(EXIT_FAILURE, "futex-FUTEX_WAKE");
    }
}
int
main(int argc, char *argv[])
{
    pid_t         childPid;
    unsigned int  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(*iaddr) * 2, PROT_READ | PROT_WRITE,
                 MAP_ANONYMOUS | MAP_SHARED, -1, 0);
```

```
    if (iaddr == MAP_FAILED)
        err(EXIT_FAILURE, "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)
        err(EXIT_FAILURE, "fork");
    if (childPid == 0) {        /* Child */
        for (unsigned int j = 0; j < nloops; j++) {
            fwait(futex1);
            printf("Child  (%jd) %u\n", (intmax_t) getpid(), j);
            fpost(futex2);
        }
        exit(EXIT_SUCCESS);
    }
    /* Parent falls through to here.  */
    for (unsigned int j = 0; j < nloops; j++) {
        fwait(futex2);
        printf("Parent (%jd) %u\n", (intmax_t) 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/locking/pi-futex.rst

? Documentation/locking/futex-requeue-pi.rst

? Documentation/locking/rt-mutex.rst

? Documentation/locking/rt-mutex-design.rst

? Documentation/robust-futex-ABI.rst

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).

Hart, D., 2009. A futex overview and update.

Hart, D. and Guniguntala, D., 2009. Requeue-PI: Making glibc Condvars PI-Aware (from proceedings of the 2009 Real-Time Linux Workshop).

Drepper, U., 2011. Futexes Are Tricky.

Futex example library, futex-*.tar.bz2.