Futexes Are Tricky

Ulrich Drepper
Red Hat, Inc.
drepper@redhat.com

November 5, 2011

Abstract

Starting with early version of the 2.5 series, the Linux kernel contains a light-weight method for process synchronization. It is used in the modern thread library implementation but is also useful when used directly. This article introduces the concept and user level code to use them.

1 Preface

The base reference for futexes has been "Fuss, Futexes and Furwocks: Fast User Level Locking in Linux" written by Franke, Russell, and Kirkwood, released in the proceedings of the 2002 OLS [1]. This document is still mostly valid. But the kernel functionality got extended and generally improved. The biggest weakness, though, is the lack of instruction on how to use futexes correctly. Rusty Russell distributes a package containing user level code (ftp://ftp.kernel.org/pub/linux/kernel/people/rusty/) but unfortunately this code is not very well documented and worse, as of this writing the code is actually incorrect.

This exemplifies that using futexes is really tricky since they provide problems even to their inventors. This document will hopefully provide correct and detailed instructions on how to use futexes. First an understanding of the kernel interface and its semantic is needed.

The following text assumes the reader is familiar with the purpose and semantics of synchronization primitives like mutex and barriers. Any book on threads will provide the necessary knowledge.

2 The Kernel Interface

The kernel interface consists mainly of one multiplexing system call:

```
long sys_futex (void *addr1, int op,
  int val1, struct timespec *timeout,
  void *addr2, int val3)
```

This prototype is actually a bit misleading, as we will later see, but it is sufficient for now. The futex itself is a variable of type `int` at the user level, pointed to by `addr1`. It has a size of 4 bytes on all platforms, 32-bit and 64-bit. The value of the variable is fully under the control of the application. No value has a specific meaning.[1]

Any memory address in regular memory (excluding something like DMA areas etc) can be used for the futex. The only requirement is that the variable is aligned at a multiple of `sizeof(int)`.

It is not obvious from the prototype, but the kernel handles the actual physical addresses of the futexes. I.e., if two processes reference a futex in a memory region they share, they will reference the same futex object. This allows the implementation of inter-process synchronization primitives.

The various actions which can be performed on a futex can be selected with the `op` parameter which can have the following values:

`FUTEX_WAIT` This operation causes the thread to be suspended in the kernel until notified. The system call returns with the value zero in this case. Before the thread is suspended the value of the futex variable is checked. If it does not have the same value as the `val1` parameter the system call immediately returns with the error `EWOULDBLOCK`.

In case the `timeout` parameter is not `NULL`, the thread is suspended only for a limited time. The `struct timespec` value specifies the number of seconds the calling thread is suspended. If the time runs out without a notification being sent, the system call returns with the error `ETIMEDOUT`.

Finally the system call can return if the thread received a signal. In this case the error is `EINTR`.

The `addr2` parameter is not used for this operation and no specific values have to be passed to the kernel.

[1] With the exception of the futex used for notification of thread termination. This is not discussed here.

FUTEX_WAKE To wake up one or more threads waiting on a futex this operation can be used. Only the `addr1`, `op`, and `val1` parameters are used. The value of the `val1` parameter is the number of threads the caller wants to wake. The type is `int`, so to wake up all waiting threads it is best to pass `INT_MAX`.

Usually the only values which are used are 1 and `INT_MAX`. Everything else makes little sense given that the list of waiters will depend on the relative execution time each thread gets and therefore cannot be foreseen in general. This means it cannot be determined from user level which threads get woken. And even if it would be possible for one situation, this is an implementation detail which might change.. Values smaller or equal to zero are invalid.

The kernel does *not* look through the list of waiters to find the highest priority thread. The normal futexes are not realtime-safe. There might be extensions in future which are, though.

Whether the woken thread gets executed right away or the thread waking up the others continues to run is an implementation detail and cannot be relied on. Especially on multi-processor systems a woken thread might return to user level before the waking thread. This is something we will investigate later a bit more.

The return value of the system call is the number of threads which have been queued and have been woken up.

FUTEX_WAKE_OP Some operations implemented using futexes require handling of more than one futex at the same time. One such example is the conditional variable implementation which needs to have an internal lock and a separate wait queue. The internal lock has to be obtained before every operation. But this can lead then to heavy context switching due to contention. Image thread A wakes up thread B which is waiting for a futex. Whenever the futex is handled an internal lock must be acquired. If thread B is scheduled before thread A released the internal lock it immediately has to go back to sleep, just now on the wait queue of the internal list (and no, in this case it is not possible to use `FUTEX_CMP_REQUEUE` as described below.

To avoid this and similar problems the `FUTEX_WAKE_OP` operation was developed. It works like `FUTEX_WAKE` but whether the wakeup actually happens depends on the result of a conditional expression involving a memory location, preceded on an operation of the same memory location. In C the operation can be expressed like this:

```
int oldval = *(int *)addr2;
*(int *)addr2 = oldval OP OPARG;
futex_wake(addr1, val1);
if (oldval CMP CMPARG)
  futex_wake(addr2, val2);
```

where `OP`, `OPARG`, `CMP`, and `CMPARG` are encoded in the `val3` parameter using

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

This might seem like a strange operation but it handles the case neither of the other wake up operations (`FUTEX_WAKE` and `FUTEX_CMP_REQUEUE`) can handle efficiently. The encoding for the operations is as follows:

Name	Value	Operation
`FUTEX_OP_SET`	0	`*addr2=OPARG`
`FUTEX_OP_ADD`	1	`*addr2+=OPARG`
`FUTEX_OP_OR`	2	`*addr2|=OPARG`
`FUTEX_OP_ANDN`	3	`*addr2&=~OPARG`
`FUTEX_OP_XOR`	4	`*addr2^=OPARG`

The comparison operations are encoded like this:

Name	Value	Operation
`FUTEX_OP_CMP_EQ`	0	`oldval==CMPARG`
`FUTEX_OP_CMP_NE`	1	`oldval!=CMPARG`
`FUTEX_OP_CMP_LT`	2	`oldval<CMPARG`
`FUTEX_OP_CMP_LE`	3	`oldval<=CMPARG`
`FUTEX_OP_CMP_GT`	4	`oldval>CMPARG`
`FUTEX_OP_CMP_GE`	5	`oldval>=CMPARG`

FUTEX_CMP_REQUEUE This operation implements a superset of the `FUTEX_WAKE` operation. It allows to wake up a given number of waiters. The additional functionality is that if there are more threads waiting than woken, they are removed from the wait queue of the futex pointer to by `addr1` and added to the wait queue of the futex pointed to by `addr2`. The number of threads treated this way can also be capped: the `timeout` parameter is misused for that. The numeric value of the pointer argument is converted to an `int` and used. We call this value here `val2`. The whole operation is only started if `val3` is still the value of the futex pointed to by `addr1`. If this is not the case anymore the system call returns with the error `EAGAIN`.

The threads moved to the second futex's wait queue can then be handled just like any other threads waiting on that futex. They can be woken individually or in batches. When the requeued thread returns there is no indication whatsoever that this requeue operation happened.

Useful values for the `val1` parameter for this operation are zero and one. `INT_MAX` is not useful since this would mean this operation behaves just like `FUTEX_WAKE`. The `val2` value is usually either one or `INT_MAX`. Using Zero makes no sense since, again, this operation would degenerate to `FUTEX_WAIT`.

The return value of the system call specifies how many threads have been woken or queued at the second futex's wait queue. The caller can determine wheter any thread has been requeued; this is the case only if the value is greater than `val1`.

`FUTEX_REQUEUE` This operation is the now obsolete predecessor of `FUTEX_CMP_REQUEUE`. It proved to be broken and unusable. No new code should use this operation, it is only kept for compatibility reasons. The difference is that `FUTEX_REQUEUE` does not support the `val3` parameter and therefore changes to the futex corresponding to the destination wait queue are not detected. This can lead to deadlocks.

> *Support for the `FUTEX_FD` operation has meanwhile been removed from the kernel because it is not really usable at all. The references to this operation in this text remain only for historical reasons.*

`FUTEX_FD` The semantics of this operation is different from the others. No operation is performed on the futex. Instead, the kernel generates a file descriptor which can then be used to refer to the futex. In addition it is possible to request asynchronous notification.

This operation requires only the `addr1` and `val1` parameter to be passed to the kernel. If the `val1` parameter is zero, the system call return value is a new file descriptor, created for the futex `addr1`. This file descriptor then can be used in `select`, `poll`, or `epoll` calls. Whenever the thread got woken or signalled the `select`/`poll`/`epoll` operation can return. The `revents` field for the file descriptor is filled with `POLLIN|POLLRDNORM` if some woke waiters of the futex. The wakeup is edge-triggered.

In case the `val1` parameter is not zero it must be the value for a valid signal. The kernel associates this signal with the returned file descriptor so that it is sent in case the thread is woken while waiting on the futex.

From these descriptions it is apparent that a fundamental detail of the futex is the wait queue in the kernel which is associated with it. Waiters are enqueued, wakers dequeue threads. These operations have to be performed atomically and more, the test of the futex value in the `FUTEX_WAIT` calls must be atomic, too. This means that the operation to wait on a futex is composed of getting the lock for the futex, checking the current value, if necessary adding the thread to the wait queue, and releasing the lock. Waking threads get the lock, then wake or requeue threads, before releasing the lock. It is important that the steps are executed in this order to guarantee that threads which go to sleep because the futex value is unchanged are going to be woken if once the futex value is changed and threads are woken. The internal locks of the futex implementation guarantee this atomicity. The following sections show how all this together allows implementing synchronization primitives.

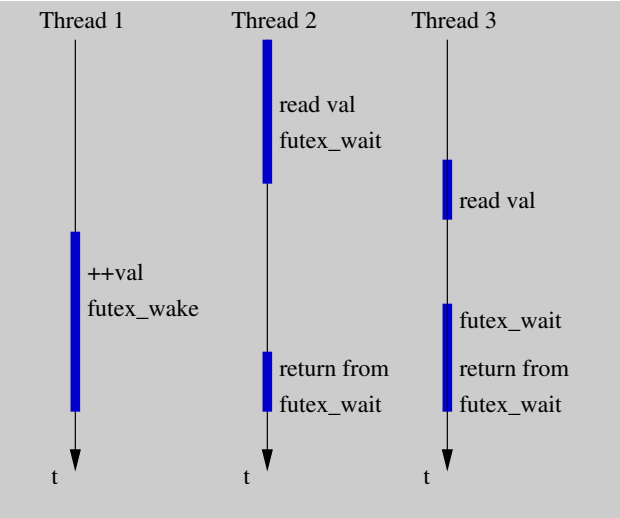
In the remainder of the text we use the interfaces listed in appendix A. The implementation of these interfaces is architecture dependent. None of the interfaces is part of the standard runtime. Programs wishing to use them probably have to provide their own implementation.

3 Why Do Futexes Work?

As an introduction we are going to examine one of the simplest possible uses of futexes. It is not really a synchronization primitive but still can be perceived as usable. We build an object that allows a thread to be notified of the arrival of a new event. The implementation could look like this.

```
class event
{
 public:
  event () : val (0) { }
  void ev_signal ()
    { ++val;
      futex_wake (&val, INT_MAX); }
  void ev_wait ()
    { futex_wait (&val, val); }
 private:
  int val;
};
```

Objects of this type can be used to synchronize arbitrarily many threads inside a process. All threads interested in synchronizing with each other need to use the same object. There can be multiple objects inside a process, which would allow synchronizing in separate clusters of threads.



This diagram represents the execution of a program with three threads. Each thread's execution is represented by a vertical line, progressing with time downward. The blue parts are those at which the thread actually has a CPU. If two or more blue lines overlap vertically the threads are executed concurrently.

In this specific example thread 2 calls `ev_wait` which reads the value of `val` and passes it to the kernel in the `futex_wait` call. This is where the thread is suspended. The value of `val` passed down is still the current value and therefore there is no need to return with `EWOULDBLOCK`. The third thread also reads the value, but is then interrupted. The value is stored in a register or in some temporary memory.

Now thread 1 calls `ev_signal` to wake up all waiters. First it increments the `val` and then calls into the kernel to wake the waiter (all of them since the parameter with the count is `INT_MAX`). At the same time as thread 1 makes the system call thread 3 also enters the kernel, to wait. After the `futex_wake` call is finished both thread 2 and 3 can resume. It is noteworthy, though, that the reason why both threads continue is different.

Thread 2 returns since it is woken by thread 1. The return value of the system call is zero. Thread 3 on the other hand did not even go to sleep. The value of `val` passed to the kernel in the third parameter is different from the value `val` has when the kernel processes the futex system call in thread 3: in the meantime thread 1 incremented `val`. Therefore thread 3 returns immediately with `EWOULDBLOCK`, independent of thread 1's wakeup call.

The experienced programmer of parallel programs will certainly have noticed a problem in the code. The use of `++val` in a multi-threaded program is not safe. This does not guarantee that all threads see consistent values. In this first example there is no real problem since the events 'increment', 'wake', and 'wait' are so weakly ordered, that using an atomic increment instruction or not does not make much of a difference.

The second simplest operation is probably mutual exclusion. The mutex implementation is essential for almost all the other mechanisms we will look into. It also explains the nuances of the futex system call we have not touched yet so we will devote some time and lines to explaining the mechanism is detail.

4 Mutex, Take 1

Be warned ahead of time that the implementation we develop in this section is not 100% kosher. We will discuss the shortfalls at the end of this section and show a possible solution in the next. This two-step process helps to further exemplify the use of futexes. Readers can try to spot the problem before it is explained.

For a mutex, it is critical that at most one thread at any time can own the mutex and that, if the mutex is free, either one or more threads are trying to lock the mutex, or the list of waiters for the mutex is empty. These requirements add quite a bit more complexity to the code. One possible implementation can look like this:

```
class mutex
{
 public:
  mutex () : val (0) { }
  void lock () {
    int c;
    while ((c = atomic_inc (val)) != 0)
      futex_wait (&val, c + 1); }
  void unlock () {
    val = 0; futex_wake (&val, 1); }
 private:
  int val;
};
```

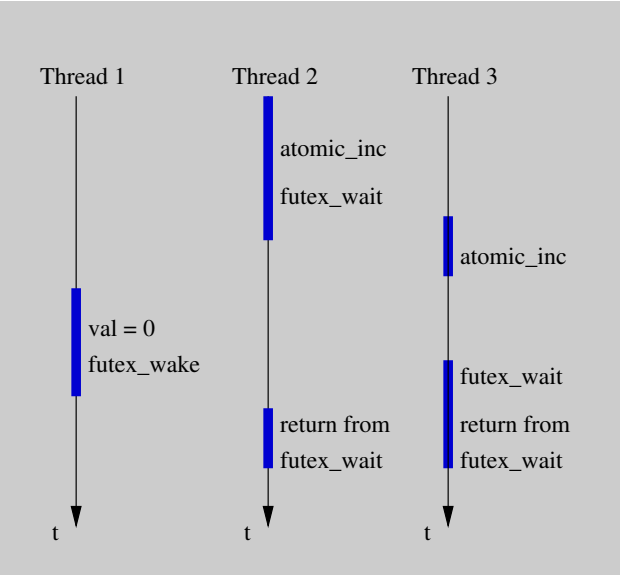
To understand the implementation we first look at the value the member `val` can have. Its initial value is zero, which means the mutex is not taken; all other values mean the mutex is taken. In the `lock` member function we see a call to `atomic_inc` which atomically increments the member `val` and then returns the *old* value. If the old value is zero the function returns. If the old value is not zero the function `futex_wait` is called. Two things are important about this: first, the call happens in a loop. We cannot guarantee that if the `futex_wait` call returns the thread will get the mutex. Instead the thread has to try locking the mutex again. Second, the value passed as the current value of the futex is the value of `val` before the `atomic_inc` plus one. The "plus one" part is important since otherwise the call would probably return right away with an `EWOULDBLOCK` error value.

Unlike in the last example code this time we did use an atomic instruction. If we would have used a simple increment like `++val` instead of the call to `atomic_inc` and two threads would execute the `lock` member function at the same time on different processors of one system, then both threads might get zero as the old value back. This can happen if the memory access is not synchronized between the CPUs and the result would be a violation of the mutex definition: more than one thread successfully called `lock` before either one called `unlock` and therefore two threads entered the critical region.

The `unlock` function is very simple. It first stores the value representing an unlock mutex. The new value must be stored atomically. We do not use a special instruction since simple load and store instructions are usually atomic. The call to `futex_wake` wakes one thread. This is different from how we used this function before when we woke up all waiters. This would be possible here as well, but it would be a waste of resources. Imagine a mutex with 100 waiters, perhaps on a multi-processor machine. Even if we would wake up all threads only one thread can lock the mutex. That means 99 threads would probably go back to sleep right away. And what is worse: since the 100 threads are distributed over all processors and all threads have to access the same `val` member, the cache line containing this value is passed from on CPU to the other and back. This is a *very* expensive operation. Therefore calling `futex_wake` with one as the second parameter is a significant optimization.

Now that we understand how the code works it is necessary to verify that the requirements on the mutex functionality are fulfilled. It is guaranteed that at most one thread can hold the mutex. If this would not be the case the `atomic_inc` function must return zero for more than one thread. This in turn is only possible when between the two `atomic_inc` calls `val` has been reset to zero, which finally means the mutex has been unlocked. Therefore this requirement is fulfilled.

The second requirement is that either the wait queue is empty, the mutex is locked, or at least one thread tries to lock the mutex. The wait queue is maintained by the kernel as part of the futex implementation. It cannot be directly observed, we have to deduce the status from the operations which have been performed. If the mutex is locked the wait queue does not matter, so we can ignore this case. This means we have to show it is not possible that if the mutex is unlocked, the wait queue is not empty, and no thread tries to lock the mutex. The attempts to lock the mutex happen in the loop in the `lock` member function. Any thread that ever tried to lock the mutex either returned from `lock` successfully (and since the mutex is unlocked, later called `unlock`) or is still in the loop. Therefore what remains to be shown is that even though a mutex got unlocked after one or more threads found it locked, at least one thread left the wait queue after the `unlock` call is finished.



The preceding diagram shows the cases we have to consider. Thread 1 holds initially the mutex. Thread 2 tries to lock it, the `atomic_inc` call returns a value other than zero, and the thread goes to sleep. There could be already other threads waiting. But once thread 1 has stored the zero value in `val` and called `futex_wake`, one of the threads on the wait queue is woken and will return to compete for the mutex. The requirement is fulfilled. The only other possibility for a thread entering the loop is that it behaves like thread 3. The `atomic_inc` call returned a nonzero value, but before the thread can be added to the wait queue thread 1 resets `val` to zero. This means thread 3 will return right away with error value `EWOULDBLOCK`. If both thread 2 and 3 are executed as indicated in this diagram it means that they both will compete for the mutex when they return from the `futex_wait` call. So in theory it would not have been necessary for thread 1 to wake thread 2 with a call to `futex_wait` since with thread 3 never being added to the wait queue the mutex requirements would still have been met. But the code in `unlock` is not clever enough to avoid unnecessary calls and in this specific case it would not be possible to avoid the wakeup since whether thread 3 is added to the wait queue or not depends on the race between thread 1 resetting `val` and thread 3 being added to the wait queue. The result need not always be the same and every time when writing synchronization primitives one must plan for the worst case.

As mentioned at the beginning of this section, the simple mutex code shown above has problems. One performance problem, and even two correctness problem.

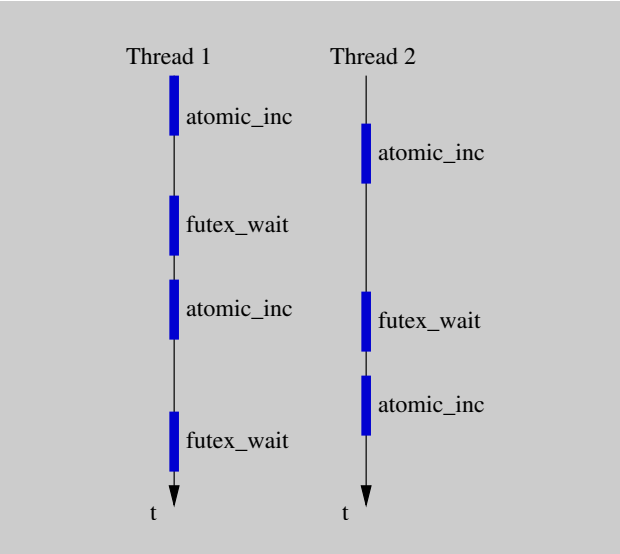
- Imagine the mutex is uncontested at all times. The `unlock` member function will still in the end always call `futex_wake` which in turn will make a system call. This can be quite costly and is in this case avoidable.

  The problem stems from the fact that the state the

mutex code keeps is very coarse grained. If `val` is zero, the mutex is unlocked. Otherwise it is locked. What we would need to do is to recognize one more state: locked and no waiters. If `unlock` is called with the futex in this state the call to `futex_wake` could be skipped.

- The first bug is quite serious in some situations but very hard to spot.[2] The loop in `lock` has the problem that between the memory read (part of the `atomic_inc` call) and the thread being added to the wait queue after the value was found to be still valid there is a sufficiently large window for causing problems.

  Consider the following diagram. After thread 1 incremented `val` it tries to put itself to sleep. But at the same time thread 2 tries to do the same, also incrementing `val`. The `futex_wait` call thread 1 does now fails with `EWOULDBLOCK`. When the system call returns `val` is incremented again. If now thread 2 calls `futex_wait` it is in the same situation: it returns with `EWOULDBLOCK` and increments `val`. This process can be continued ad infinitum.



  It might seem that such a behavior is rare and could be discounted. But this is not the case. First, the futex implementation in the kernel is serializing uses of a specific futex. Since in our example the threads all use the same futex this means all the futex calls are serialized. On single processor systems the possibility that a thread gets interrupted right after the `atomic_inc` call is pretty low, but it is still possible. On multi processor system the threads running on other processors can make the critical `atomic_inc` calls anytime. The more processors are involved trying to lock the same mutex the higher the possibility, especially if locking the mutex is a big part of the work. In one case a real world application running on a four processor machine got sped up eight to ten times by fixing this problem. The extremely expensive cache line transfer necessary for the atomic accesses make this bug very costly.

- The second bug has to do with the nature of recording waiters. New waiters unconditionally increment the `val`. But this variable has a finite size. On all the interesting systems this means after 2^{32} increments we are back to zero and magically the variable is free. This is not as esoteric as it seems since it does not require 2^{32} threads. Every time the `futex_wait` call returns but the mutex has not been unlocked the variable is incremented. I.e., it is in theory possible for one single thread to overflow the counter. The remaining question is: when can `futex_wait` return erroneously? One example is the first bug above. But there is also a way which cannot be avoided. In the introduction it was explained that the `FUTEX_WAIT` operation is interrupted if the thread received a signal. This certainly can happen in any program and it can happen a lot.

  For this reason it is in most cases necessary to avoid boundless increments. This usually comes at a price so one might want to examine whether this bug is for real in the given specific situation one wants to use the futex in or not.

5 Mutex, Take 2

A generally usable mutex implementation must at least fix the two bugs identified in the last section. Ideally it should also address the first point of critique. To summarize:

- the livelocks caused by the unconditional change of the futex variable must be avoided;
- the futex value must not overflow;
- in case it is known no threads wait on the mutex the `futex_wake` call should be avoided.

To represent the states we need at least three distinct values and since we don't want to overflow the variable we keep it at that. The following code uses therefore the following convention:

0	unlocked
1	locked, no waiters
2	locked, one or more waiters

Restricting the mutex variable to three values while still supporting multi processor machines means we cannot use the `atomic_inc` function anymore. Instead we use a

[2] This bug was present in some form for many months in the NPTL [2] implementation. It showed mainly up as mysterious slowdowns and occasional bursts of CPU usage.

function which is available on many platforms with one single instruction: a compare-and-exchange instruction `cmpxchg` (see appendix A for more details). Architectures which do not provide such an instruction can be supported by emulating it (e.g., with load lock/store conditional). The resulting code looks like this:

```
class mutex2
{
 public:
  mutex () : val (0) { }
  void lock () {
    int c;
    if ((c = cmpxchg (val, 0, 1)) != 0)
      do {
        if (c == 2
            || cmpxchg (val, 1, 2) != 0)
          futex_wait (&val, 2);
      } while ((c = cmpxchg (val, 0, 2))
               != 0);
  }
  void unlock () {
    if (atomic_dec (val) != 1) {
      val = 0;
      futex_wake (&val, 1);
    }
  }
 private:
  int val;
};
```

This code is certainly all but obvious at first sight. We will dissect it in a minute. First let us take a look at the performance. The fast path used if no thread contention exists is very important and needs to be optimized for.

		mutex	mutex2
lock	atomic op	1	1
	futex syscall	0	0
unlock	atomic op	0	1
	futex syscall	1	0

We can see that there is no difference for `lock` which needs in any case one atomic operation. It might be, that this still translates to a slowdown since the atomic increment operation is sometimes faster than a compare-and-exchange operation. This depends on the CPU details. The important case here is the cost for the `unlock` function. We traded one system call for an atomic operation. This is almost always a good choice, especially here since the futex system call needs atomic instructions itself. The benefits of this change is substantial. What about the cost for the contended case?

		mutex	mutex2
lock	atomic op	$1+1$	$\frac{2}{3}+\frac{1}{2}$
	futex syscall	$1+1$	$1+1$
unlock	atomic op	0	1
	futex syscall	1	1

These results look worse for the new code and in fact, `mutex2` is indeed slower than the `mutex` code for contended mutexes. But this is the price we have to pay for correctness. The shortcut in the conditional inside the loop in `lock` makes computing the cost a bit more difficult. If there are already waiters for the mutex the code avoids the expensive `cmpxchg` instruction. In the cost table the two stacked numbers represent these different costs. In case there are already waiters use the upper number, otherwise the lower number. The $+N$ part in the fields represents the additional cost for the function call which has to be paid if the `futex_wait` system call returns but the thread cannot get the mutex and is going back to sleep.

We see significantly higher costs for the `lock` function and slightly higher costs for `unlock`. We make the same number of system calls in all cases,but the `lock` function makes 2 to 3 times as many atomic operations; `unlock` has one more atomic operation to make. All of `lock`'s additional cost are attributed to correcting the bug. The additional `unlock` cost is a consequence of optimizing the case of an uncontested mutex. It has been found useful to do this since mutexes are also used in single threaded applications and even in multi-threaded applications many mutex operations find the mutex unlocked. If this code is found to be correct the additional cost is therefore well spent. We will now go into details of the code to show how it works and why it is correct.

First we will look at `unlock`. Not only because it is simpler, also because the `lock` code depends on its implementation. When discussing the costs we already mentioned that the `atomic_dec` call is used to optimize the code path in case the mutex is uncontested, i.e., there are no waiters. According to the table with the state values this state is represented by 1. Therefore the return value of `atomic_dec` in case there is no waiter is 1. We skip the `futex_wake` system call in this case which would be unnecessary since the wait queue for the futex is empty. In case the state value is 2 we make the system call to wake a thread if there is any. We wake only one thread; as with the `mutex` code there is no need to wake more than one since all but one thread probably would have to go back to sleep.

Now on to `lock`. The intent for the first `cmpxchg` call is to distinguish the uncontested case from the more complicated and slower cases. If the mutex is unlocked (status value 0) it is marked as locked with no waiters by changing the value to 1. This is all done by this one instruction. Success can be tested for by comparing the old

value, returned by `cmpxchg` with 0. In case of a match we are done.

It gets complicated only if the mutex is already locked. We have two cases to distinguish: there is no waiter and there is (perhaps) one or more waiters. The "perhaps" might be irritating, it will become clearer later. If there is no waiter so far we have to indicate that now there is one. The state value for this is 2. This means we have to change the value from 1 to 2 which is exactly what the second `cmpxchg` does. We know that this function call will do nothing in case we already have waiters which is why we have the shortcut for `c == 2`.[3] Then it is time to suspend the thread. There is only one more case to handle: in case the second `cmpxchg` failed since the mutex is freed we should not make the system call. Instead we can try to get the mutex right away. In all other cases the `futex_wait` call will suspend the thread. Note that the expected value for the futex is unconditionally 2.

Once the `futex_wait` call returns or we did not make the call, another attempt to take the mutex has to be made. This is now the most non-obvious operation: we try to change the state from unlocked (i.e., 0) to locked. But we must use the 'locked with possible waiters' state 2 and not the simple 'locked' state 1. Why? The answer is: because we do not know any better. When we come to this point we cannot say with 100% certainty that there is not already a waiter. Since being wrong in guessing sooner or later means running into a deadlock we have to err on the safe side. Here this means we have to mark the mutex as possibly locked multiple times. The "perhaps" in the initial description should have become clear. The consequence is a possible unnecessary call to `futex_wake` in `unlock`.

Showing that the code is correct more formally is possible but a lot of work. We just outline the key points here. First, the `lock` function only ever returns after successfully locking the mutex. The locking thread itself sets the futex value to 1. Other threads, while waiting, might set it to 2. But only the `unlock` function resets the value to 0. This ensure the actual locking. Waking up possible lockers is guaranteed by them setting the futex value to 2 which causes the `unlock` function to wake one caller. All threads which are "in flight", attempting to lock the mutex, when it is unlocked, do not block in the kernel since the futex value is changed to 0 during unlock and `lock` always passes 2 as the second parameter to `futex_wait`.

But what about the livelock situation mentioned in the last section? Can this happen here? The answer is no. If the mutex is locked, there is at most one more change of the futex value: the first thread calling `lock` changes it from 1 to 2. All other threads calling `lock` recognize that the value is set to 2 and will not change it. This is the important difference. The `cmpxchg` operation might be a bit more expensive than the `atomic_inc` but it is necessary. It might be possible in some situations to avoid the initial `cmpxchg` but this is not the case the code should be optimized for.

[3] Remember: the || operator in C/C++ will avoid evaluating the right-hand side expression if the left-hand size expression is true.

6 Mutex, Take 3

We are not yet done optimizing the code, at least not for some architectures. The repeated `cmpxchg` operations in the locking code are necessary to ensure the value 2 is really written into the memory location before the system call. For many architectures this is as good as it gets. But the IA-32 and AMD64/Intel64 architectures have one more ace in their sleeves: they have an atomic `xchg` operation (without the `cmp`). This comes in handy in our situations.

```
class mutex3
{
 public:
  mutex () : val (0) { }
  void lock () {
    int c;
    if ((c = cmpxchg (val, 0, 1)) != 0) {
      if (c != 2)
        c = xchg (val, 2);
      while (c != 0) {
        futex_wait (&val, 2);
        c = xchg (val, 2);
      }
    }
  }
  void unlock () {
    if (atomic_dec (val) != 1) {
      val = 0;
      futex_wake (&val, 1);
    }
  }
 private:
  int val;
};
```

From the description in the last section it should be clear that the code does exactly the same. The `unlock` code is unchanged, and so is the fast path of the `lock` function. The slow path of the `lock` function is now using `xchg`. The two `cmpxchg` instructions in the old code were needed because the value of the variable might change at the same time and we had to make sure we wrote the value 2 in the memory location. Now we do it unconditionally. By using the result of the `cmpxchg` operation we can save a `xchg` call in the first round. This brings us to the following costs for the contended case:

		mutex2	mutex3
lock	atomic op	$\frac{2}{3} + \frac{1}{2}$	$\frac{1}{2} + 1$
	futex syscall	$1 + 1$	$1 + 1$
unlock	atomic op	1	1
	futex syscall	1	1

The new code has only advantages and in case of contended mutexes it can make a big difference. The difference between executing one or two atomic operations on the same memory location on multiple processors at the same time is big. The actual runtime of the application might not be directly improved but the system load goes down and the memory performance improves.

The drawback of this new code is that it is not universally implementable in this form. If the architecture requires an atomic `xchg` operation to be implemented in terms of `cmpxchg` the benefits is zero, or less. Many modern architectures fall into this category. Beside the already mentioned IA-32, AMD64, and Intel64 architectures it is possible to efficiently implement `xchg` on architectures which use load lock/store conditional.

7 Inter-Process

The POSIX thread interface defines synchronization interfaces not only for the user inside processes. They can also be used between processes and futexes make this possible to implement.

One requirement of an inter-process synchronization primitive is that it is a) position independent and b) has no references/pointers to any object in any of the virtual address space. This means wait queues have to be kept somewhere else, in the case of futexes this happens in the kernel. Looking at the `mutex2` definition we see that the only state necessary for the mutex implementation is the private member `val`. This means to use the a `mutex2` object for inter-process synchronization we only have to create some shared memory segment and use the placement syntax when creating the mutex object.

```
int fd = shm_open ("/global-mutex",
                   O_RDWR, 0);
void *p = mmap (NULL, sizeof (mutex2),
                PROT_READ|PROT_WRITE,
                MAP_SHARED, fd, 0);
mutex2 *m = new (p) mutex2 ();
```

This code segment can be used in arbitrarily many processes on the same machine and they all will use the same mutex; the kernel knows that all the virtual addresses are mapped to the same physical memory address and futexes are identified by their physical address. Inter-process mutexes of this kind are a *very* much better synchronization than filesystem-based approaches like lock files. Lock files have the advantage, though, that they can synchronize on different machines. Pick your poison wisely.

8 Optimizing Wakeup

One of the most damaging effects of running a multi-threaded application on a multi-processor machine is repeated transfer of memory cache lines from one processor to the other (a.k.a. cache line ping-pong). This happens when threads running on different processors try to access the same memory address. This is a natural occurrence when implementing synchronization primitives; if only one thread would ever try to access the mutex it would not be needed at all.

One particularly bad case with respect to cache line ping-pong is the `pthread_cond_broadcast` function of the POSIX thread library. It has the potential to wake up large numbers of threads. But the threads cannot right away return from the calls to `pthread_cond_wait` or `pthread_cond_timedwait`. Instead the API requires that the POSIX mutex associated with the conditional variable is locked first. All waiting threads must use the same mutex. If we start all threads with a call to `futex_wake` and a sufficiently high number as the second parameter, the threads might be spread out to all available processors and they hammer on the memory used for the mutex.[4] This means the cache line(s) used for the representation of the mutex are copied from one processor cache to the other. All this sending of notification and copying is very expensive. And usually all but one thread have to go back to sleep anyway since the mutex can belong to only one of the woken threads.

The Linux kernel futex implementation provides a special interface for this situation. Instead of waking all threads we wake only one. But we cannot leave the other threads on the wait queue they were on before since this would defeat the purpose of waking up all threads. Instead we can move the content (or part of it) of one wait queue to another wait queue where the threads then can be woken one by one.

In the example of the `pthread_cond_broadcast` function the implementation can move all the threads to the wait queue of the futex belonging to the mutex used with the conditional variable. The `pthread_unlock` call the user code has to issue after the return of the function call which caused the thread to be added to the wait queue of the conditional variable already wakes waiters one by one. Therefore the `pthread_cond_broadcast` code can move all woken waiters to the wait queue of the mutex. Result: one by one wakeup, no cache line ping-pong, and no more going back to sleep immediately for all but one thread.

[4] This is a simplification. In any implementation all threads would first hammer on the memory of the conditional variable. But the result is the same.

The wakeup code in the pthread_cond_broadcast function would look something like this:

```
futex_requeue (cond_futex, 1, MAX_INT,
               mutex_futex, cond_val)
```

This call would move all but one of the waiters in the wait queue of the conditional variable to the wait queue of the mutex. The cond_val parameter helps to detect whether the conditional variable has changed since the initiation of the requeue operation. In this case nothing is done and the caller has to handle the new situation appropriately. It is important to ensure that the implementation of pthread_mutex_unlock really tries to wake up a thread from the waitqueue once the directly woken thread calls this function. This might be a problem since there have been no previous pthread_mutex_lock calls. Implementing all this requires a lot of tricky code.

The FUTEX_CMP_REQUEUE operation used to implement futex_requeue is only useful in special cases. Its usefulness might not become apparent on uni-processor machines and maybe even small multi-processor machines. But as soon as the threads are running on more than four processors[5] the negative effects of the cache line ping-pong are so huge that using this operation shows measurable and sometimes dramatic effects.

9 Waiting on Multiple Events

In some situations it is useful to wait on more than one event at once. For instance, a thread could perform two different tasks, both need protection by a mutex, depending on the availability on the mutex. Whichever task's mutex becomes available first is started. There is no such interface in the standard POSIX thread library. So this is a good example for an extension made by the users. The futex authors had this in mind when they introduced the FUTEX_FD operation.

A user program would call futex_fd to get one or more file descriptors for futexes. Then this file descriptor, together with possibly many others representing real files or sockets or the like, gets passed to select, poll, or epoll. This seem to help a great deal.

There is one problem with this approach. The futex_wait interface's second parameter is used to detect races. If a second thread changes the state of the synchronization object between the time of the last test before the futex_wait call and the time the kernel adds the thread to the wait queue, this is detected. The futex_wait call returns with the error EWOULDBLOCK. But no such provision exists for the interface to the futex using the file descriptor. None of the three interfaces, select, poll, and epoll, supports passing such information down.

This limitation dramatically reduces the usefulness of the FUTEX_FD operation. No synchronization interface which depends on exact wakeup can be used with this interface. For instance, the mutex2 code falls into this category. Only if a wakeup event can safely be missed is FUTEX_FD useful.

10 Other Synchronization Primitives

Most non-trivial programs using threads or multiple processes need some more complicated synchronization primitives than just mutexes. Those part of the standard POSIX thread library (and therefore deemed generally useful) are:

- barriers
- conditional variables
- read/write mutexes
- semaphores

All primitives but the simple semaphore have in common that they need some internal variables to represent the state. Modifying the state must happen as part of a critical region so each of the synchronization objects also has a mutex in it. The actual waiting for the barrier/conditional variable/RW lock does happen with the help of a different futex, also part of the synchronization object. In some cases there can even be more than these two futexes, the system does not impose a limit. When designing such a solution it is important, though, to keep the limitations imposed by cache lines in mind.

As a simple example consider the barrier. The object needs an internal counter which keeps track of the number of still needed waiters. This state is protected by a futex and those threads, which arrived before the last one, will need to go to sleep. So the interface for a barrier could look like this:

```
class barrier
{
 public:
  barrier (unsigned int needed)
    : mutex (), event (0),
      still_needed (needed),
      initial_needed (needed) { }
  wait () {
    lock.lock ();
    if (still_needed-- > 1) {
      unsigned int ev = event;
      lock.unlock ();
```

[5] This is an experience value for IA-32.

```
      do
        futex_wait (event, ev);
      while (event == ev);
    } else {
      ++event;
      still_needed = initial_needed;
      futex_wake (event, INT_MAX);
      lock.unlock ();
    }
  }
 private:
  mutex3 lock;
  unsigned int event;
  unsigned int still_needed;
  unsigned int initial_needed;
};
```

The first member variable `lock` is the mutex, as defined before. The second data member `event` is the second futex. Its value changes whenever the last waiter arrives and a new round begins. The other two values are the current and initial count of waiters needed. The implementation for this class is straight-forward with the tricky mutex implementation already in place. Note that we can simply use `futex_wake` to wake all the threads. Even though this might mean we start many threads and possibly spread them to many processors, this is different from the situation discussed in the last section. The crucial difference is that upon return from the system call the threads do *not* have to get another lock. This is at least not the case in the code related to barriers.

Not all code is simple, though. The conditional variable implementation is very complicated and way beyond the scope of this little introduction.

In section 7 we said that the synchronization object should not contain any memory references/pointers to make them usable for inter-process synchronization. This is no hard requirement for the use of futexes. If it is known that an object is never used inter-process and the use of pointers provides an advantage in the implementation, by all means, use pointers.

A Library Functions

`atomic_dec(var)` The variable `var` will be atomically decremented and the old value is returned.

`atomic_inc(var)` The variable `var` will be atomically incremented and the old value is returned.

`cmpxchg(var, old, new)` The content of the variable `var` will be replaced with `new` if the current value is `old`. Regardless, the current value of `var` before the operation is returned.

`futex_fd(futex, signal)` Create a file descriptor for `futex` which can be used in `select`, `poll`, and `epoll` calls. If `signal` is not zero and the value for a valid signal, the kernel will send this signal in case the thread gets woken while waiting.

`futex_requeue(from, nwake, nrequeue, to, fromval)` The call wakes up at most `nwake` threads from the wait queue of `from`. If there are more threads left after that, up to `nrequeue` threads are moved to the wait queue of `to`. An error is returned and no wait queue is modified if the value of the futex `from` is not `fromval`.

`futex_wait(futex, val)` If the value of the `int` variable `futex` is still `val`, wait until woken by a signal or a call to `futex_wake`.

`futex_wake(futex, nwake)` Wake up at most `nwake` threads from the wait queue of `futex`.

B References

[1] Hubertus Franke, Rusty Russell, and Matthew Kirkwood, *Fuss, Futexes and Furwocks: Fast Userlevel Locking in Linux*, Proceedings of the 2002 Ottawa Linux Summit, 2002.

[2] Ulrich Drepper, Ingo Molnar, *The Native POSIX Thread Library for Linux*, Red Hat, Inc., 2003.

C Revision History

2003-10-12 First draft.

2003-10-17 Typos. Version 0.3.

2003-10-29 Better English. Patches by Todd Lewis `todd.lewis@gs.com` and Alexandre Oliva `aoliva@redhat.com`. Version 0.4.

2004-02-22 Add `mutex3` description. Version 0.6.

2004-04-21 Typo fix. Version 0.7.

2004-06-21 More typo fixes. Version 0.8.

2004-06-27 Describe `FUTEX_CMP_REQUEUE`. Version 1.0.

2004-12-13 Fix little mistake in `cmpxchg` description (reported by Neil Conway). Version 1.2.

2005-12-11 Describe `FUTEX_WAKE_OP`. Version 1.3.

2008-01-29 And `FUTEX_OP_ANDN` description. Version 1.4.

2009-08-01 Fix a few typos. Version 1.5.

2011-11-05 Fix a typo. Version 1.6.