

Fuss, Futexes and Furwocks: Fast Userlevel Locking in Linux

Hubertus Franke

IBM Thomas J. Watson Research Center

frankeh@watson.ibm.com

Rusty Russell

IBM Linux Technology Center

rusty@rustcorp.com.au

Matthew Kirkwood

matthew@hairy.beasts.org

Abstract

Fast userlevel locking is an alternative locking mechanism to the typically heavy weight kernel approaches such as fcntl locking and System V semaphores. Here, multiple processes communicate locking state through shared memory regions and atomic operations. Kernel involvement is only necessary when there is contention on a lock, in order to perform queueing and scheduling functions. In this paper we discuss the issues related to user level locking by following the history of ideas and the code to the current day. We present the efficacy of "futexes" through benchmarks, both synthetic and through adaptations to existing databases. We conclude by presenting the potential future directions of the "futex" interface.

1 Introduction

LinuxTM¹ has seen significant growth as a server operating system and has been successfully deployed in enterprise environments for Web, file and print serving. With the deployment of Version 2.4, Linux has seen a tremendous boost in scalability and robustness that

makes it now feasible to deploy even more demanding enterprise applications such as high end databases, business intelligence software and application servers. As a result, whole enterprise business suites and middleware such as SAPTM, WebsphereTM, Oracle, DB2TM², etc., are now available for Linux.

For these enterprise applications to run efficiently on Linux, or on any other operating system for that matter, the OS must provide the proper abstractions and services. Enterprise applications and applications suites are increasingly built as multi process / multi-threaded applications. Multi-threaded applications can take better advantage of SMP hardware, while multiple processes allows for higher degrees of fault tolerance, i.e., a single process abort does not necessarily bring the entire application down. Furthermore, applications suites are often a collection of multiple independent subsystems.

Despite their functional separation, the processes representing these subsystems often must communicate with each other and share state amongst each other. Examples of this are database systems, which typically maintain shared I/O buffers in user space. The buffers

¹Linux is a trademark of Linus Torvalds

²All third party trademarks are the property of their respective owners.

are concurrently accessed by various database engines and prefetching processes.

Access to such shared state must be properly synchronized through either exclusive or shared locks. Exclusive locks allow only one party access to the protected entity, while shared locks allow multiple reader – single writer semantics. Synchronization implies a shared state, indicating that a particular resource is available or busy, and a means to wait for its availability. The latter one can either be accomplished through busy-waiting or through a explicit / implicit call to the scheduler.

In traditional UNIX™³ systems, System V IPC (inter process communication) such as *semaphores*, *msgqueues*, *sockets* and the file locking mechanism (**flock()**) are the basic mechanisms for two processes to synchronize. These mechanisms expose an opaque handle to a kernel object that naturally provides the shared state and atomic operations in the kernel. Services must be requested through system calls (e.g., **semop()**). The drawback of this approach is that every lock access requires a system call. When locks have low contention rates, the system call can constitute a significant overhead.

One solution to this problem is to deploy user level locking, which avoids some of the overhead associated with purely kernel-based locking mechanisms. It relies on a user level lock located in a shared memory region and modified through atomic operations to indicate the lock status. Only the contended case requires kernel intervention. The exact behavior and the obtainable performance are directly affected by how and when the kernel services are invoked. The idea described here is not new. Some of the foundation of this paper are described in [4], [7] and [6]. In [2] the impact of locking on JVM performance is discussed.

In this paper we are describing a particular fast user level locking mechanism called *futexes* that was developed in the context of the Linux operating system. It consists of two parts, the user library and a kernel service that has been integrated into the Linux kernel distribution version 2.5.7.

The paper is organized as followed. In section 2 we describe the basic behavioral and functional requirements of a user level locking mechanism. In section 3 we describe some of the earlier approaches that led to the current design of *futexes* and the *futexes* themselves. In section 4 we provide a performance assessment on a synthetic and a database benchmark. In section 5 we elaborate on current and future efforts and in 6 we conclude.

2 Requirements

In this section we are stating some of the requirements of a fast userlevel locking mechanism that we derived as part of this work and that were posted to us as requirements by middleware providers.

There are various behavioral requirements that need to be considered. Most center around the fairness of the locking scheme and the lock release policy. In a **fair** locking scheme the lock is granted in the order it was requested, i.e., it is handed over to the longest waiting task. This can have negative impact on throughput due to the increased number of context switches. At the same time it can lead to the so called **convoy problem**. Since, the locks are granted in the order of request arrival, they all proceed at the speed of the slowest process, slowing down all waiting processes. A common solution to the convoy problem has been to mark the lock available upon release, wake all waiting processes and have them recontend for the lock. This is referred to as **random fairness**,

³UNIX is a trademark of The Open Group

although higher priority tasks will usually have an advantage over lower priority ones. However, this also leads to the well known **thundering herd problem**. Despite this, it can work quite well on uni-processor systems if the first task to wake releases the lock before being preempted or scheduled, allowing the second herd member to obtain the lock, etc. It works less spectacularly on SMP. To avoid this problem, one should only wake up one waiting task upon lock release. Marking the lock available as part of releasing it, gives the releasing task the opportunity to reacquire the lock immediately again, if so desired, and avoid unnecessary context switches and the convoy problem. Some refer to these as **greedy**, as the running task has the highest probability of reacquiring the lock if the lock is hot. However, this can lead to starvation. Hence, the basic mechanisms must enable both fair locking, random locking and greedy or convoy avoidance locking (short ca-locking). Another requirement is to enable spin locking, i.e., have an application spin for the availability of the lock for some user specified time (or until granted) before giving up and resolving to block in the kernel for its availability. Hence an application has the choice to either (a) block waiting to be notified for the lock to be released, or (b) yield the processor until the thread is rescheduled and then the lock is tried to be acquired again, or (c) spin consuming CPU cycles until the lock is released.

With respect to performance, there are basically two overriding goals:

- avoid system calls if possible, as system calls typically consume several hundred instructions.
- avoid unnecessary context switches: context switches lead to overhead associated with TLB invalidations etc.

Hence, in fast userlevel locking, we first distinguish between the uncontended and the contended case. The uncontended case should be efficient and should avoid system calls by all means. In the contended case we are willing to perform a system call to block in the kernel.

Avoiding system calls in the uncontended case requires a shared state in user space accessible to all participating processes/task. This shared state, referred to as the *user lock*, indicates the status of the lock, i.e., whether the lock is held or not and whether there are waiting tasks or not. This is in contrast to the System V IPC mechanisms which merely exports a handle to the user, and performs all operations in the kernel.

The user lock is located in a shared memory region that was create via **shmat ()** or **mmap ()**. As a result, it can be located at different virtual addresses in different address spaces. In the uncontended case, the application atomically changes the lock status word without entering into the kernel. Hence, atomic operations such as **atomic_inc ()**, **atomic_dec**, **cmpxchg ()**, and **test_and_set ()** are necessary in user space. In the contended case, the application needs to wait for the release of the lock or needs to wake up a waiting task in the case of an unlock operation. In order to wait in the kernel, a *kernel object* is required, that has *waiting queues* associated with it. The waiting queues provide the queuing and scheduling interactions. Of course, the aforementioned IPC mechanisms can be used for this purpose. However, these objects still imply a heavy weight object that requires a priori allocation and often does not precisely provide the required functionality. Another alternative that is commonly deployed are *spinlocks* where the task spins on the availability of the user lock until granted. To avoid too many cpu cycles being wasted, the task yields the processor occasionally.

It is desirable to have the user lock be handle-free. In other words instead of handling an opaque *kernel handle*, requiring initialization and cross process global handles, it is desirable to address locks directly through their virtual address. As a consequence, kernel objects can be allocated dynamically and on demand, rather than apriori.

A lock, though addressed by a virtual address, can be identified conceptually through its *global lock identity*, which we define by the memory object backing the virtual address and the offset within that object. We notate this by the tuple [B,O]. Three fundamental memory types can be distinguished that represent B: (a) anonymous memory, (b) shared memory segment, and (c) memory mapped files. While (b) and (c) can be used between multiple processes, (a) can only be used between threads of the same process. Utilizing the virtual address of the lock as the kernel handle also provides for an integrated access mechanism that ties the virtual address automatically with its kernel object.

Despite the atomic manipulation of the user level lock word, race conditions can still exist as the sequence of lock word manipulation and system calls is not atomic. This has to be resolved properly within the kernel to avoid deadlock and improper functioning.

Another requirement is that fast user level locking should be simple enough to provide the basic foundation to efficiently enable more complicated synchronization constructs, e.g. semaphores, rwlocks, blocking locks, or spin versions of these, pthread mutexes, DB latches. It should also allow for a clean separation of the blocking requirements towards the kernel, so that the blocking only has to be implemented with a small set of different constructs. This allows for extending the use of the basic primitives without kernel modifica-

tions. Of interest is the implementation of mutex, semaphores and multiple reader/single writer locks.

Finally, a solution needs to be found that enables the recovery of “dead” locks. We define unrecoverable locks as those that have been acquired by a process and the process terminates without releasing the lock. There are no convenient means for the kernel or for the other processes to determine which locks are currently held by a particular process, as lock acquisition can be achieved through user memory manipulation. Registering the process’s “pid” after lock acquisition is not enough as both operations are not atomic. If the process dies before it can register its pid or if it cleared its pid and before being able to release the lock, the lock is unrecoverable. A protocol based solution to this problem is described in [1]. We have not addressed this problem in our prototypes yet.

3 Linux Fast User level Locking: History and Implementations

Having stated the requirements in the previous section, we now proceed to describe the basic general implementation issues. For the purpose of this discussion we define a general opaque datatype `ulock_t` to represent the userlevel lock. At a minimum it requires a status word.

```
typedef struct ulock_t {
    long status;
} ulock_t;
```

We assume that a shared memory region has been allocated either through `shmat()` or through `mmap()` and that any user locks are allocated into this region. Again note, that the addresses of the same lock need not be the same across all participating address spaces.

The basic semaphore functions **UP()** and **DOWN()** can be implemented as follows.

```
static inline int
usema_down(uunlock_t *unlock)
{
    if (!__unlock_down(unlock))
        return 0;
    return sys_unlock_wait(unlock);
}

static inline int
usema_up(uunlock_t *unlock)
{
    if (!__unlock_up(unlock))
        return 0;
    return sys_unlock_wakeup(unlock);
}
```

The `__unlock_down()` and `__unlock_up()` provide the atomic increment and decrement operations on the lock status word. A non positive count (status) indicates that the lock is not available. In addition, a negative count *could* indicate the number of waiting tasks in the kernel. If a contention is detected, i.e. `(unlock->status <= 0)`, the kernel is invoked through the `sys_*` functions to either wait on the wait queue associated with `unlock` or release a blocking task from said waitqueue.

All counting is performed on the lock word and race conditions resulting from the non-atomicity of the lock word must be resolved in the kernel. Due to such race conditions, a lock can receive a wakeup before the waiting process had a chance to enqueue itself into the kernel wait queue. We describe below how various implementation resolved this race condition as part of the kernel service.

One early design suggested was the explicit allocation of a kernel object and the export of the kernel object address as the handle. The kernel object was comprised of a wait queue and a unique security signature. On every wait or

wakeup call, the signature would be verified to ensure that the handle passed indeed was referring to a valid kernel object. The disadvantages of this approach have been mentioned in section 2, namely that a handle needs to be stored in `unlock_t` and that explicit allocation and deallocation of the kernel object are required. Furthermore, security is limited to the length of the key and hypothetically could be guessed.

Another prototype implementation, known as *ulocks* [3], implements general user semaphores with both fair and convoy avoidance wakeup policy. Mutual exclusive locks are regarded as a subset of the user semaphores. The prototype also provides multiple reader/single writer locks (rwlocks). The user lock object `unlock_t` consists of a lock word and an integer indicating the required number of kernel wait queues. User semaphores and exclusive locks are implemented with one kernel wait queue and multiple reader/single writer locks are implemented with two kernel wait queues.

This implementation separates the lock word from the kernel wait queues and other kernel objects, i.e., the lock word is never accessed from the kernel on the time critical wait and wakeup code path. Hence the state of the lock and the number of waiting tasks in the kernel is all recorded in the lock word. For exclusive locks, standard counting as described in the general `unlock_t` discussion, is implemented. As with general semaphores, a positive number indicates the number of times the semaphore can be acquired, "0" and less indicates that the lock is busy, while the absolute of a negative number indicates the number of waiting tasks in the kernel.

The "premature" wakeup call is handled by implementing the kernel internal wait-queues using kernel semaphores (`struct semaphore`) which are initialized with a

value 0. A premature wakeup call, i.e. no pending waiter yet, simply increases the kernel semaphore's count to 1. Once the pending wait arrives it simply decrements the count back to 0 and exits the system call without waiting in the kernel. All the wait queues (kernel semaphores) associated with a user lock are encapsulated in a single kernel object.

In the rwlocks case, the lock word is split into three fields: write locked (1 bit), writes waiting (15 bits), readers (16 bits). If write locked, the **readers** indicate the number of tasks waiting to read the lock, if not write locked, it indicates the numbers of tasks that have acquired read access to the lock. Writers are blocking on a first kernel wait queue, while readers are blocking on a second kernel wait queue associated with a ulock. To wakeup multiple pending read requests, the number of task to be woken up is passed through the system call interface.

To implement rwlocks and ca-locks, atomic compare and exchange support is required. Unfortunately on older 386 platforms that is not the case.

The kernel routines must identify the kernel object that is associated with the user lock. Since the lock can be placed at different virtual addresses in different processes, a lookup has to be performed. In the common fast lookup, the virtual address of the user lock and the address space are hashed to a kernel object. If no hash entry exists, the proper global identity $[B, O]$ of the lock must be established. For this we first scan the calling process's vma list for the vma containing the lock word and its offset. The global identity is then looked up in a second hash table that links global identities with their associated kernel object. If no kernel object exists for this global identity, one is allocated, initialized and added to the hash functions. The **close()** function associated with a shared region holding kernel objects is inter-

cepted, so that kernel objects are deleted and the hash tables are cleaned up, once all attached processes have detached from the shared region.

While this implementation provides for all the requirements, the kernel infrastructure of multiple hash tables and lookups was deemed too heavy. In addition, the requirement for compare and exchange is also seen to be restrictive.

3.1 Futexes

With several independent implementations [8, 9, 10] in existence, the time seemed right in early 2002 to attempt to combine the best elements of each to produce the minimum useful subset for insertion into the experimental Linux kernel series.

There are three key points of the original futex implementation which was added to the 2.5.7 kernel:

1. We use a unique identifier for each futex (which can be shared across different address spaces, so may have different virtual addresses in each): this identifier is the "struct page" pointer and the offset within that page. We increment the reference count on the page so it cannot be swapped out while the process is sleeping.
2. The structure indicating which futex the process is sleeping on is placed in a hash table, and is created upon entry to the futex syscalls on the process's kernel stack.
3. The compression of "fast userspace mutex" into "futex" gave a simple unique identifier to the section of code and the function names used.

3.1.1 The 2.5.7 Implementation

The initial implementation which was judged a sufficient basis for kernel inclusion used a single two-argument system call, “**sys_futex(struct futex *, int op)**”. The first argument was the address of the futex, and the second was the operation, used to further demultiplex the system call and insulate the implementation somewhat from the problems of system call number allocation. The latter is especially important as the system call is expanded as new operations are required. The two valid op numbers for this implementation were **FUTEX_UP** and **FUTEX_DOWN**.

The algorithm was simple, the file *linux/kernel/futex.c* containing 140 code lines, and 233 in total.

1. The user address was checked for alignment and that it did not overlap a page boundary.
2. The page is pinned: this involves looking up the address in the process’s address space to find the appropriate “**struct page ***”, and incrementing its reference count so it cannot be swapped out.
3. The “**struct page ***” and offset within the page are added, and that result hashed using the recently introduced fast multiplicative hashing routines [11], to give a hash bucket in the futex hash table.
4. The “op” argument is then examined. If it is **FUTEX_DOWN** then:
 - (a) The process is marked *INTERRUPTIBLE*, meaning it is ready to sleep.
 - (b) A “**struct futex_q**” is chained to the tail of the hash bucket determined in step 3: the tail is chosen

to give FIFO ordering for wakeups. This structure contains a pointer to the process and the “**struct page ***” and offset which identify the futex uniquely.

- (c) The page is mapped into low memory (if it is a high memory page), and an atomic decrement of the futex address is attempted,⁴ then unmapped again. If this does not decrement the counter to zero, we check for signals (setting the error to **EINTR** and going to the next step), schedule, and then repeat this step.
- (d) Otherwise, we now have the futex, or have received a signal, so we mark this process *RUNNING*, unlink ourselves from the hash table, and wake the next waiter if there is one, and return **0** or **-EINTR**. We have to wake another process so that it decrements the futex to -1 to indicate that it is waiting (in the case where we have the futex), or to avoid the race where a signal came in just as we were woken up to get the futex (in the case where a signal was received).

5. If the op argument was **FUTEX_UP**:

- (a) Map the page into low memory if it is in a high memory page
- (b) Set the count of the futex to one (“available”).
- (c) Unmap the page if it was mapped from high memory

⁴We do not even attempt to decrement the address if it is already negative, to avoid potential wraparound. We do the decrement even if the counter is zero, as “-1” indicates we are sleeping and hence has different semantics than 0.

- (d) Search the hash table for the first “**struct futex_q**” associated with this futex, and wake up that process.
6. Otherwise, if the `op` argument is anything else, set the error to `EINVAL`.
7. Unpin the page.

While there are several subtleties in this implementation, it gives a second major advantage over System V semaphores: there are no explicit limits on how many futexes you can create, nor can one futex user “starve” other users of futexes. This is because the futex is merely a memory location like any other until the `sys_futex` syscall is entered, and each process can only do one `sys_futex` syscall at a time, so we are limited to pinning one page per process into memory, at worst.

3.1.2 What about Read-Write Locks?

We considered an implementation of “`FUTEX_READ_DOWN`” et. al, which would be similar to the simple mutual exclusion locks, but before adding these to the kernel, Paul Mackerras suggested a design for creating read/write lock in userspace by using two futexes and a count: *fast userspace read/write locks*, or *furwocks*. This implementation provides the benchmark for any kernel-based implementation to beat to justify its inclusion as a first-class primitive, which can be done by adding new valid “`op`” values. A comparison with the integrated approach chosen by `ulocks` is provided in Section 4.

3.1.3 Problems with the 2.5.7 Implementation

Once the first implementation entered the mainstream experimental kernel, it drew the attention of a much wider audience. In particular those concerned with implementing POSIX(tm)⁵ threads, and attention also returned to the fairness issue.

- There is no straightforward way to implement the `pthread_cond_timedwait` primitive: this operation requires a timeout, but using a timer is difficult as these must not interfere with their use by any other code.
- The `pthread_cond_broadcast` primitive requires every process sleeping to be woken up, which does not fit well with the 2.5.7 implementation, where a process only exits the kernel when the futex has been successfully obtained or a signal is received.
- For N:M threading, such as the Next Generation Posix Threads project [5] an asynchronous interface for finding out about the futex is required, since a single process (containing multiple threads) might be interested in more than one futex.
- Starvation occurs in the following situation: a single process which immediately drops and then immediately competes for the lock will regain it before any woken process will.

With these limitations brought to light, we searched for another design which would be flexible enough to cater for these diverse needs. After various implementation attempts and discussions we settled on a variation of `atomic_compare_and_swap` primitive, with

⁵POSIX is a trademark of the IEEE Inc.

the atomicity guaranteed by passing the expected value into the kernel for checking. ? To do this, two new “op” values replaced the operations above, and the system call was changed to two additional arguments, “int val” and “struct timespec *reltime”.

FUTEX_WAIT: Similar to the previous FUTEX_DOWN, except that the looping and manipulation of the counter is left to userspace. This works as follows:

1. Set the process state to *INTERRUPTIBLE*, and place “struct futex_q” into the hash table as before.
2. Map the page into low memory (if in high memory).
3. Read the futex value.
4. Unmap the page (if mapped at step 2).
5. If the value read at step 3 is not equal to the “val” argument provided to the system call, set the return to **EWOULDBLOCK**.
6. Otherwise, sleep for the time indicated by the “reltime” argument, or indefinitely if that is NULL.
 - (a) If we timed out, set the return value to **ETIMEDOUT**.
 - (b) Otherwise, if there is a signal pending, set the return value to **EINTR**.
7. Try to remove our “**struct futex_q**” from the hash table: if we were already removed, return 0 (success) unconditionally, as this means we were woken up, otherwise return the error code specified above.

FUTEX_WAKE: This is similar to the previous **FUTEX_UP**, except that it does not

alter the futex value, it simply wakes one (or more) processes. The number of processes to wake is controlled by the “int val” parameter, and the return value for the system call is the number of processes actually woken and removed from the hash table.

FUTEX_AWAIT: This is proposed as an asynchronous operation to notify the process via a SIGIO-style mechanism when the value changes. The exact method has not yet been settled (see future work in Section 5).

This new primitive is only slightly slower than the previous one,⁶ in that the time between waking the process and that process attempting to claim the lock has increased (as the lock claim is done in userspace on return from the FUTEX_WAKE syscall), and if the process has to attempt the lock multiple times before success, each attempt will be accompanied by a syscall, rather than the syscall claiming the lock itself.

On the other hand, the following can be implemented entirely in the userspace library:

1. All the POSIX style locks, including pthread_cond_broadcast (which requires the “wake all” operation) and pthread_cond_timedwait (which requires the timeout argument). One of the authors (Rusty) has implemented a “non-pthreads” demonstration library which does exactly this.
2. Read-write locks in a single word, on architectures which support cmpxchg-style primitives.

⁶About 1.5% on a low-contention tdbtorture, 3.5% on a high-contention tdbtorture

3. FIFO wakeup, where fairness is guaranteed to anyone waiting (see 3.1.4).

Finally, it is worthwhile pointing out that the kernel implementation requires exactly the same number of lines as the previous implementation: 233.

3.1.4 FIFO Queueing

The naive implementation of “up” does the following:

1. Atomically set the futex to 1 (“available”) and record the previous value.
2. If the previous value was negative, invoke `sys_futex` to wake up a waiter.

Now, there is the potential for another process to claim the futex (without entering the kernel at all) between these two steps: the process woken at step 2 will then fail, and go back to sleep. As long as this does not lead to starvation, this unfairness is usually tolerable, given the performance improvements shown in Section 4

There is one particular case where starvation is a real problem which must be avoided. A process which is holding the lock for extended periods and wishes to “give way” if others are waiting cannot simply do “`futex_up(); futex_down();`”, as it will always win the lock back before any other processes.

Hence one of us (Hubertus) added the concept of “`futex_up_fair()`”, where the futex is set to an extremely negative number (“passed”), instead of 1 (“available”). This looks like a “contended” case to the fast userspace “`futex_down()`” path, as it is negative, but indicates to any process after a successful return from the `FUTEX_WAIT` call that

the futex has been passed directly, and no further action (other than resetting the value to -1) is required to claim it.

4 Performance Evaluation

In this section we assess the performance of the current implementation. We start out with a synthetic benchmark and continue with a modified database benchmark.

4.1 MicroBenchmark: UlockFlex

Ulockflex is a synthetic benchmark designed to ensure the integrity and measure the performance of locking primitives. In a run, *Ulockflex* allocates a finite set (typically one) of global shared regions (shmat or mmap’ed files) and a specified number of user locks which are assigned to the shared region in a round robin fashion. It then clones a specified number of tasks either as threads or as processes and assigns each task to one particular lock in a round robin fashion. Each cloned task, in a tight loop, computes two random numbers *nlht* and *lht*, acquires its assigned lock, does some work of lock hold time *lht*, releases the lock, does some more work of non-lock hold time *nlht* and repeats the loop. The mean lock hold time *lht(mean)* and non-lock hold times *nlht(mean)* are input parameters. *lht* and *nlht* are determined as random numbers over a uniform distribution in the interval [0.5..1.5] of their respective mean. The tool reports total cumulative throughput (as in number of iterations through the loop). It also reports the coefficient of variance of the per task throughput. A higher coefficient indicates the potential for starvation. A small coefficient indicates fairness over the period of execution. A data structure associated with each lock is updated after obtaining the lock and verified before releasing the lock, thus allowing for integrity checks.

In the following we evaluate the performance of various user locking primitives that were built on the basics of the `futex` and the `ulocks` implementations. We consider the basic two wakeup policies for both `futexes` and `ulocks`, i.e. fair wakeup and regular wakeup (i.e. convoy avoidance), yielding the 4 cases `futex_fair`, `futex`, `ulocks_fair` and `ulocks`. For these cases we also consider a spinning lock acquisition in that the task tries to acquire the lock for 3 μsecs before giving up and blocking in the kernel, yielding the 4 cases of `futex_fair(spin,3)`, `futex(spin,3)`, `ulocks_fair(spin,3)` and `ulocks(spin,3)`. For reference we also provide the measurements for a locking mechanism build on System V semaphores, i.e., each lock request results in a system call. This variant is denoted as `sysv`, resulting in 9 overall locking primitives being evaluated.

All experiments were performed on a dual Pentium-III 500 MHz, 256MB system. A data point was obtained by running `ulockflex` for 10 seconds with a minimum of 10 runs or until a 95% confidence interval was achieved.

In the first experiment we determine the basic overhead of the locking mechanisms. For this we run with one task, one lock and $nlht == lht == 0$. Note that in this case all user locking mechanisms never have to enter into the kernel. Performance is reported as % efficiency of a run without lock invocations. The `sysv` was 25.1% efficient, while all 8 user level locking cases fell within 84.6% and 87.9%. When the $(nlht + lht)$ was increased to 10 μsecs , the efficiency of `sysv` was still only 82.2%, while those of the user level locks ranged from 98.9% to 99.1%.

When executing this setup with two tasks and two locks the efficiency of `sysv` drops to 18.3% from 25.1% indicating a hot lock in the kernel. At the same time the user level primitives

all remain in the same range, as expected. The same effect can be described as follows. With this setup we would expect twice the throughput performance as compared to the 1 task, 1 lock setup. Indeed, for all user primitives the scalability observed is between 1.99 and 2.02, while `sysv` only shows a scalability of 1.51.

In the next set of experiments we fixed the total loop execution time $nlht + lht$ to 10 μsecs , however we changed the individual components. Let $(nlht, lht)$ denote a configuration. Four configurations are observed: (0,10), (5,5), (7,3), (9,1). The (0,10) represents the highly contended case, while (9,1) represents a significantly less contended case. The exact contention is determined by the number of tasks accessing a shared lock. Contention numbers reported are all measured against the fair locking version of `ulocks` in a separate run. The contention measurement does not introduce any significant overhead.

Figures 1.5 show the comparison of the 9 locking primitives for the four configurations under various task counts (2,3,4,100,1000). The percentage improvements for each configuration and task count over the `sysv` base number for that configuration are reported in Table 1 for the fair `futexes` and `ulocks` without and with spinning (3 μsecs) and in Table 2 for the regular `futexes` and `ulocks`.

The overall qualitative assessment of the results presented in these figures and tables is as follows. First comparing the fair locking mechanisms, fair `ulocks`, in general, have an advantage over fair `futexes`. Furthermore, fair `futexes` perform worse than `sysv` for high contention scenarios. Only in the high task count numbers do fair `futexes` outperform (substantially) `sysv` and fair `ulocks`. Spinning only showed some decent improvement in the low contention cases, as expected. For the regular versions (ca-locks), both `futexes` and `ulocks`

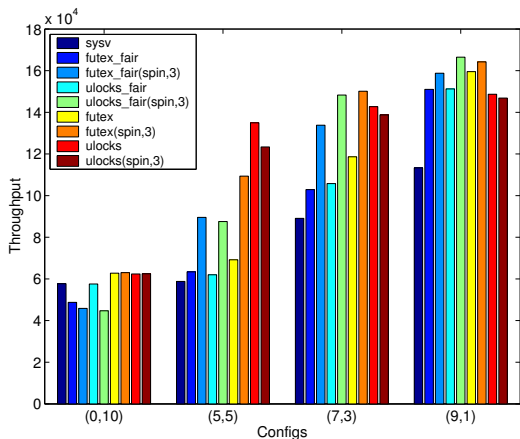


Figure 1: Throughput for various lock types for 2 tasks, 1 lock and 4 configurations

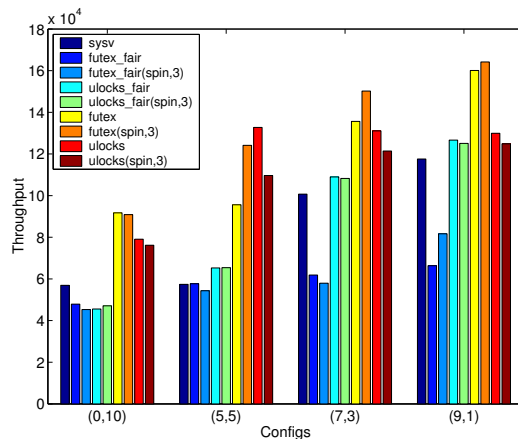


Figure 3: Throughput for various lock types for 4 tasks, 1 lock and 4 configurations

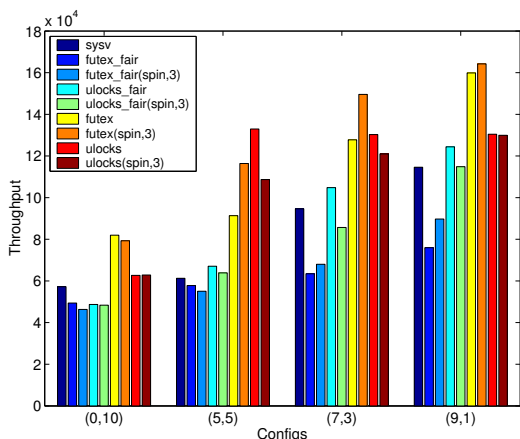


Figure 2: Throughput for various lock types for 3 tasks, 1 lock and 4 configurations

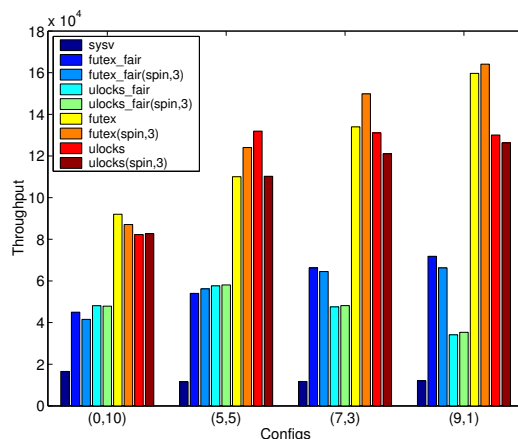


Figure 4: Throughput for various lock types for 100 tasks, 1 lock and 4 configurations

always outperform the *sysv* version. The general tendency is for *ulocks* to achieve their performance at the **(5, 5)** configuration with little additional benefits. Though *futexes* in general lack the *ulock* performance at the **(5, 5)** configuration, they outperform *ulocks* at the **(7, 3)** and the **(9, 1)** configurations. In contrast to *futexes*, spinning for *ulocks* does not help.

Figure 1 shows the results for 2 tasks competing for 1 lock under four contention scenarios. The lock contention for the 4 configurations were 100%, 97.8%, 41.7% and 13.1%. The

lock contention observed for Figure 2.. 5 are all above 99.8%.

We now turn our attention to the multiple reader/single writer (*rwlock*) lock primitives. To recall, *furwocks* implement the *rwlock* functionality ontop of two regular *futexes*, while *ulocks* implement them directly in the interface through atomic compare and exchange manipulation of the lock word. *Ulockflex* allows the specification of a **share-level** for *rwlocks*. This translates into the probability of a task requesting a read lock instead of a write lock while iterating through the tight loop.

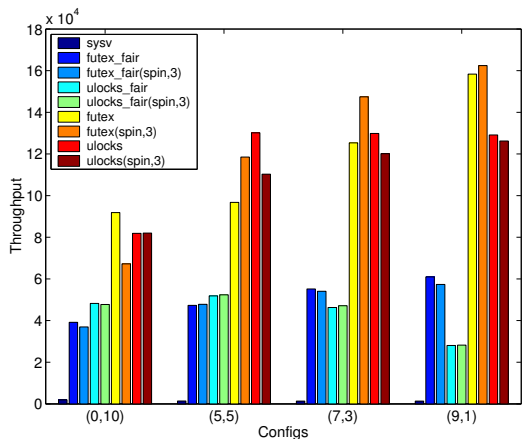


Figure 5: Throughput for various lock types for 1000 tasks, 1 lock and 4 configurations

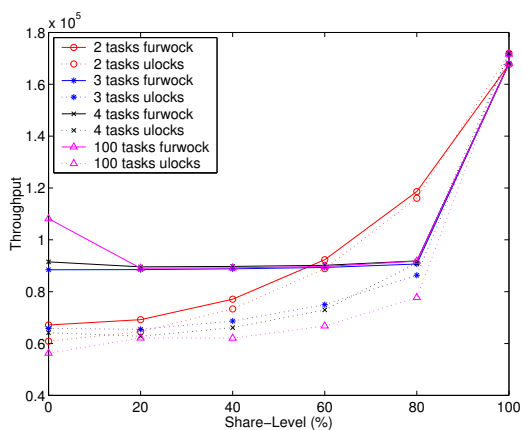


Figure 6: Throughput of furwocks and shared ulocks for (2,3,4,100) tasks competing for a single lock under different read share ratios

Figure 6 shows the achieved throughput of furwocks and shared ulocks for 2, 3, 4 and 100 tasks competing for a single lock under different read share ratios. The general observation is that the furwocks (solid lines) outperform the ulocks (dashed lines) for their respective task numbers. In general the lower the share level and/or the higher the task numbers the better the improvements that can be achieved with furwocks over shared ulocks. Only in the 100% share-level (only read accesses) do shared ulocks outperform furwocks by $\tilde{2}$ -3%.

We now analyze the fairness of the user locking. We monitor the global fairness by computing the coefficient of variance *coeff* of the per task throughput. Note this should not be compared with the fair locking itself. The *coeff* of *sysv* is typically below 0.01. Only the 1000 task case showed a *coeff* of 9.1, indicating that tasks did not all properly get started. The *coeff* for fair futexes and fair ulocks for small task numbers (2,3,4) is in general below 0.01 (as expected). For large task number (100,1000), the *coeff* remains very low for futexes, while ulocks experience a *coeff* as high as 1.10. For furwocks, the general observation is that the *coeff* is less than 0.16 in both furwocks and shared ulocks. Only for the 100 task case does the *coeff* reach 0.45. Overall the mean of *coeff* for all scenarios is 0.068 for furwocks and 0.054 for shared ulocks. In general we can state that at these level of contention, global starvation is not a problem.

We now turn our attention to the degree of local fairness for the ca-locks. We do this by investigating how many times a task is capable of reacquiring the lock before some other task locks it. To do so, we examine a high contention case of 100 tasks and the (9,1) configuration. The kernel lock and the fair futexes showed perfect fairness, 99.99% of the task could never reacquire its lock without losing it to some other task. The fair ulocks only 92.1% failed to reacquire, 3.6% was able to grab the lock twice in a row and 0.4% three times. The maximum times a lock was able to be reacquired was 1034 times. For futexes these numbers are 79.0%, 21.0% and maximum of 575 and for ulocks they are 82.4%, 17.54% and maximum of 751. To some degree it confirms that futexes and ulocks have a higher degree of instant reacquisition, however this analysis fails to shed more light on why futexes are so much better than ulocks.

4.2 TDB Torture Results

The Trivial DataBase (TDB) is a simple hash-chain-based on-disk database used by SAMBA and other projects to store persistent internal data. It has a similar interface to the classic dbm library, but allows multiple readers and writers and is less than 2000 lines long. TDB normally uses fcntl locks: we replaced these with futex locks in a special part of the memory-mapped file. We also examined an implementation using "spin then yield" locks, which try to get the lock 1000 times before calling yield() to let other processes schedule.

tdbtorture is one of the standard test programs which comes with TDB: we simplified it to eliminate the cleanup traversal which it normally performs, resulting in a benchmark which forks 6 processes, each of which does 200000 random search/add/delete/traverse operations.

To examine behavior under high contention, we created a database with only one hash chain, giving only two locks (there is one lock for the free records chain). For the low contention case, we used 4096 chains (there is still some contention on the allocation lock). For the no contention case, we used a single process, rather than 6. The results shown in Table 3 were obtained on a 2-processor 350MHz Pentium II.

It is interesting that the fcntl locks have different scaling properties than futexes: they actually do much worse under the low contention case, possibly because the number of locks the kernel has to keep track of increases.

Another point to make here is the simplicity of the transformation from fcntl locks to futexes within TDB: the modification took no longer than five minutes to someone familiar with the code.

5 Current and Future Directions

Currently we are evaluating an asynchronous wait extension to the futex subsystem. The requirement for this arises for the necessity to support global POSIX mutexes in thread packages. In particular, we are working with the NGPT (next generation pthreads) team to derive specific requirements for building global POSIX mutexes over futexes. Doing so provides the benefit that in the uncontended case, no kernel interactions are required. However, NGPT supports a $M : N$ threading model, i.e., M user level threads are executed over N tasks. Conceptually, the N tasks provide virtual processors on which the M user threads are executing.

When a user level thread, executing on one of these N tasks, needs to block on a futex, it should not block the task, as this task provides the virtual processing. Instead only the user thread should be descheduled by the thread manager of the NGPT system. Nevertheless, a **waitobj** must be attached to the waitqueue in the kernel, indicating that a user thread is waiting on a particular futex and that the task group needs a notification wrt to the continuation on that futex. Once the thread manager receives the notification it can reschedule the previously blocked user thread.

For this we provide an additional operator **AFUTEX_WAIT** to the **sys_futex** system call. Its task is to append a **waitobj** to the futex's kernel waitqueue and continue. Compared to the synchronous calls described in Section 3, this **waitobj** can not be allocated on the stack and must be allocated and deallocated dynamically. Dynamic allocations have the disadvantage that the **waitobjs** must be freed even during an irregular program exit. It further poses a denial of service attack threat in that a malicious applications can continuously call **sys_futex(AFUTEX_WAIT)**. We are

contemplating various solutions to this problem.

The general solutions seem to convert to the usage of a `/dev/futex` device to control resource consumption. The first solution is to allocate a file descriptor `fd` from the `/dev/futex` “device” for each outstanding asynchronous `waitobj`. Conveniently these descriptors should be “pooled” to avoid the constant opening and closing of the device. The private data of the file would simply be the `waitobj`. Upon completion a SIGIO is sent to the application. The advantage of this approach is that the denial of service attack is naturally limited to the file limits imposed on a process. Furthermore, on program death, all `waitobjs` still enqueued can be easily dequeued. The disadvantage is that this approach can significantly pollute the “fd” space. Another solution proposed has been to open only one `fd`, but allow multiple `waitobj` allocations for this `fd`. This approach removes the fd space pollution issue but requires an additional tuning parameter for how many outstanding `waitobjs` should be allowed per fd. It also requires proper resource management of the `waitobjs` in the kernel. At this point no definite decisions has been reached on which direction to proceed.

The question of priorities in futexes has been raised: the current implementation is strictly FIFO order. The use of nice level is almost certainly too restrictive, so some other priority method would be required. Expanding the system call to add a priority argument is possible, if there were demonstrated application advantage.

6 Conclusion

In this paper we described a fast userlevel locking mechanism, called *futexes*, that were integrated into the Linux 2.5 development ker-

nel. We outlined the various requirements for such a package, described previous various solutions and the current futex package. In the performance section we showed, that futexes can provide significant performance advantages over standard System V IPC semaphores in all cases studies.

7 Acknowledgements

Ulrich Drepper (for feedback about current POSIX threads and glibc requirements), Paul Mackerras (for furwocks and many ideas on alternate implementations), Peter Waechtler and Bill Abt for their feedback on asynchronous notifications.

References

- [1] Philip Bohannon and et. al. Recoverable User-Level Mutual Exclusion. In *Proc. 7th IEEE Symposium on Parallel and Distributed Systems*, October 1995.
- [2] Robert Dimpsey, Rajiv Arora, and Kean Kuiper. Java Server Performance: A case study of building efficient, scalable JVMs. *IBM Systems Journal*, 39(1):151–174, 2000.
- [3] Hubertus Franke. Ulocks: Fast Userlevel Locking. Available at <http://lse.sourceforge.net>.
- [4] John M. Mellor-Crummey and Michael L. Scott. Scalable Reader-Writer Synchronization for Shared Memory Multiprocessors. *ACM Transactions on Computer Systems*, 9(1):21–65, February 1991.
- [5] NGPT: Next Generation Pthreads. Available at <http://oss.software.ibm.com/pthreads>.

- [6] Michael Scott and William N. Scherer III. Scalable Queue-Based Spin Locks with Timeouts. In *Proc. 11th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP'01*, 2001.
- [7] Robert W. Wisniewski, Leonidas I. Kontothanassis, and Michael Scott. High Performance Synchronization Algorithms for Multiprogrammed Multiprocessors. In *Proc. 5th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP'95*, 1995.
- [8] Message-ID: <Pine.LNX.4.33.0201071902070.5064-101000@sphinx.mythic-beasts.com>.
- [9] Message-ID: <20020211143841.A1674@elinux01.watson.ibm.com>.
- [10] Message-ID: <E16gRe3-0006ak-00@wagner.rustcorp.com.au>.
- [11] Message-ID: <20020106183417.L10326@holomorphy.com>.

Conf	no-spin		spin	
	futex	ulock	futex	ulock
2 tasks				
(0,10)	-15.5	-0.7	-20.5	-22.9
(5,5)	7.9	4.6	52.4	47.7
(7,3)	15.5	18.7	50.2	66.4
(9,1)	33.2	33.1	40.1	46.5
3 tasks				
(0,10)	-13.7	-15.2	-19.1	-15.9
(5,5)	-5.7	8.9	-10.1	3.8
(7,3)	-33.0	11.0	-28.2	-9.2
(9,1)	-33.7	7.5	-21.7	-0.7
4 tasks				
(0,10)	-15.8	-20.0	-20.4	-17.5
(5,5)	0.6	13.3	-5.3	13.5
(7,3)	-38.6	8.0	-42.5	7.3
(9,1)	-43.6	7.7	-30.6	6.4
100 tasks				
(0,10)	172.3	190.8	151.4	189.5
(5,5)	367.6	393.9	386.4	397.6
(7,3)	464.0	300.5	449.0	305.5
(9,1)	495.7	180.3	449.1	190.0
1000 tasks				
(0,10)	1900.4	2343.9	1787.2	2317.9
(5,5)	3363.7	3752.5	3403.7	3792.1
(7,3)	3972.5	3295.2	3891.1	3357.3
(9,1)	4393.7	1971.5	4127.7	1985.3

Table 1: Percentage improvement of Fair locking (spinning and non-spinning) over the base `sysv` throughput

Conf	no-spin		spin	
	futex	ulock	futex	ulock
2 tasks				
(0,10)	8.8	7.6	9.3	7.8
(5,5)	17.7	127.8	86.0	108.2
(7,3)	33.2	60.1	68.5	55.7
(9,1)	40.8	30.9	44.9	29.3
3 tasks				
(0,10)	43.2	9.0	38.5	9.3
(5,5)	49.1	116.0	89.9	76.5
(7,3)	35.0	38.0	58.0	28.1
(9,1)	39.5	12.8	43.3	12.3
4 tasks				
(0,10)	61.2	38.8	59.7	33.7
(5,5)	66.6	130.5	116.3	90.5
(7,3)	34.7	29.9	49.1	20.3
(9,1)	36.1	10.5	39.6	6.2
100 tasks				
(0,10)	456.8	397.1	426.9	399.7
(5,5)	852.3	1030.2	973.4	844.5
(7,3)	1040.4	1003.9	1175.2	919.5
(9,1)	1223.7	967.7	1260.4	936.5
1000 tasks				
(0,10)	4591.7	4047.9	3333.1	4055.2
(5,5)	6989.5	9570.0	8583.8	8095.9
(7,3)	9149.7	9427.1	10781.5	8714.6
(9,1)	11569.6	9437.7	11869.9	9223.3

Table 2: Percentage improvement of regular (ca) locking (spinning and non-spinning) over the base *sysv* throughput

Locktype	Contention Level		
	High	Low	None
FCNTL	1003.69	1482.08	76.4
SPIN	751.18	431.42	67.6
FUTEX	593.00	111.45	41.5

Table 3: Completion times (secs) of *tdbtorture* runs with different contention rates and different lock implementations