Deterministic Futexes Revisited

Alexander Zuepke, Robert Kaiser
RheinMain University of Applied Sciences, Wiesbaden, Germany
Email: first.last@hs-rm.de

***Abstract*—Fast User Space Mutexes (Futexes) in Linux are a lightweight way to implement thread synchronization objects like mutexes and condition variables. Futexes handle the uncontended case in user space and rely on the operating system only for suspension and wake-up on contention. However, the current futex implementation in Linux is unsuitable for hard real-time systems due to its unbounded worst case execution time (WCET).**

Based upon the ideas from our previous work presented at OSPERT in 2013 which addressed this problem, this paper presents an improved design for *Deterministic Futexes* which shows a logarithmic upper bound of the worst case execution time (WCET) and covers more futex use cases. The implementation targets microkernels or statically configured real-time operating systems.

I. Introduction

Support for Fast User Space Mutexes (Futexes) was introduced in Linux in 2002 [1] with the Native POSIX Thread Library (NPTL). Futexes allow to implement various POSIX-compliant high level synchronization objects such as mutexes, condition variables, semaphores, readers/writer locks, barriers, or one-time initializers with low overhead in the system's C library in user space. One major design goal of futexes was to reduce any system call overhead for these locking objects where possible, thus the implementation uses atomic modifications to handle uncontended locking and unlocking entirely in user space, while a generic system call-based mechanism is used to suspend and wake threads in the kernel on lock contention. Basically, a futex is a 32-bit integer variable in user space, representing a certain type of lock and its value is modified by a type-specific locking protocol [2].

Similar approaches where the kernel is entered only on contention are used by *Critical Sections* in Microsoft Windows [3] and *Benaphores* in BeOS [4].

We give a short introduction to futexes using a simple mutex implementation as example: in an integer variable, let bit 0 represent the locked state of the mutex, while bit 1 indicates contention. The unlocked mutex is represented by the value 0x0. A thread can lock and unlock the mutex by atomically changing the lock value from 0x0 to 0x1 and vice versa using a *Compare-and-Swap (CAS)* or *Load-Linked/Store-Conditional (LL/SC)* operation.

A lock operation on an already locked mutex atomically changes the value from 0x1 to 0x3 to indicate contention and then invokes a `FUTEX_WAIT` system call to suspend the calling thread until the lock becomes available again. Symmetrically, when the current lock-holder sees contention during an unlock operation, it atomically clears the locked bit in the futex value and calls the `FUTEX_WAKE` system call to wake a blocked thread which then acquires the lock by atomically setting bit 0 again. On contention, `FUTEX_WAIT` enqueues the thread on a *wait queue* which holds blocked threads referring to the same or a different user space futex. For wake-up, `FUTEX_WAKE` searches the wait queue and wakes up matching threads, if any.

The last important operation on futexes is `FUTEX_REQUEUE` to prevent *thundering herd effects* [5] when signalling condition variables: instead of waking up all threads and letting them compete to lock the associated mutex, this system call wakes only one thread and moves any remaining blocked threads from the wait queue associated to the condition variable to the mutex' one.

By design, futexes impose no restrictions on the number of user space variables used for futexes or on the number of threads blocked in a wait queue. This flexibility makes the concept very attractive and led to its recent adoption by other operating systems [6]–[8].

However, being designed for best case scenarios, the current futex *implementation* in Linux has drawbacks which make it unsuitable for hard real-time operating systems:

- *Hash table with shared wait queues:* Linux hashes the futex user space address and groups threads with the same hash value into a *shared wait queue*. This can lead to an unbounded worst case execution time (WCET) when, due to hash collisions, many unrelated threads are kept in the same hash bucket.
- *Linked lists:* Linux implements wait queues using *priority-sorted linked lists*, which show $\mathcal{O}(n)$ search time in shared wait queues and $\mathcal{O}(p)$ insertion time, for n threads and p priority levels.
- *Not preemptive:* When waking up or requeuing a large number of threads, the Linux implementation is not preemptive. Again, this can lead to an unbounded WCET.

In previous work [9], we presented a solution which tackles these problems by using a dedicated kernel-internal wait queue for each futex. To let the kernel look-up the wait queue, we placed the ID of the first waiting thread next to the futex value in user space. The solution then utilized $\mathcal{O}(1)$ insertion and deletion time of linked lists to bound the WCET. However, the solution in [9] supported only FIFO ordering in the wait queues, so it does not fulfill the POSIX requirement to wake up threads in priority order [10].

In this paper, we present an improved futex implementation with the following properties:

- dedicated wait queues for each futex,
- arbitrary ordering in the wait queues,

- bounded $\mathcal{O}(\log n)$ worst case execution time in the kernel for all futex operations targeting a single thread,
- preemptible implementation of futex operations which wake up or requeue all threads, and
- no dependency on dynamic memory allocation.

The rest of this paper is organized as follows: Section II describes all futex operations in detail and defines requirements for determinism and reliability. Section III presents our new approach. We discuss our new approach and compare it with the current Linux implementation and our previous approach in Section IV and we conclude in Section V.

II. Fast User Space Mutexes and Condition Variables

A. Terminology

Before we discuss the futex operations, we define the terminology used in the rest of this paper: a *process* is an instance of a computer program executing in an *address space*. A process comprises one or more *threads*. Threads can be independently scheduled on different processors at the same time. Different processes have their own distinct address space, but processes can share parts of their address spaces via *shared memory segments*. A shared memory segment is usually *mapped* at different virtual addresses in each address space. A *waiting* or *blocked* thread suspends execution until the thread is *woken up* or *unblocked* again.

B. Futex Operations in User Space

Here, we briefly present the user space parts of a futex-based mutex and condition variable implementation to help understanding the corresponding kernel parts. The mutex protocol extends the one shown in Section I and uses different kernel operations. Note that the presented user space implementation is simplified for ease of understanding. An actual user space implementation will usually be more complex, as the calls also have to handle asynchronous signals, thread cancellation, etc., but the interaction with the kernel side of the presented futex implementation remains the same. The presented futex API also deviates from the existing Linux API in that the handling of an arbitrary number of `count` threads is reduced to the two most common use cases, *one* or *all*. This helps to bound the WCET, as we will explain later.

Mutex: For a mutex, the futex value comprises two pieces of information: the thread ID (`TID`) of the current lock holder or 0 if the mutex is free, and a *waiters* bit if the mutex has contention. Also both user space and the kernel need to understand this mutex protocol.

`mutex_lock` first tries to lock a mutex by atomically changing the futex value from 0 to `TID`. If the mutex is already locked, `mutex_lock` atomically sets the waiters bit in the futex value to indicate contention, then calls `futex_lock` to suspend itself on the current futex value. The `futex_lock` operation in the kernel checks the futex value again and tries to either acquire the mutex for the caller if it is free, or, if not, atomically sets the waiters bit in the futex value and suspends the calling thread. On successful return from `futex_lock`, the calling thread is the new lock owner.

Conversely, `mutex_unlock` tries to unlock the mutex by atomically changing the futex value from `TID` to 0. If this fails (the waiters bit is set), `mutex_unlock` calls `futex_unlock` in the kernel. If no threads are waiting, `futex_unlock` sets the futex value to 0, or wakes up the next waiting thread and makes it the new lock owner by updating the `TID` in the futex value. `futex_unlock` sets the waiters bit as well if other threads are still waiting.

Condition Variable: For a condition variable, the futex value represents a counter that is incremented on each wake-up operation. The kernel does not need to know the exact protocol. When doing any operation on a condition variable, we assume the caller also has the associated mutex locked [10].

`cond_wait` reads the condition variable's counter value, unlocks the associated mutex, and then calls `futex_wait` to block with an optional timeout on the condition variable if the current counter value still matches the previously read value. Additionally, `cond_wait` provides the mutex object to later requeue to as well.

`cond_signal` and `cond_broadcast` increment the counter and call `futex_requeue` to requeue either one or all blocked threads from the condition variable's wait queue to the mutex' wait queue. In case the caller has not locked the mutex before, `futex_requeue` checks whether the associated mutex is unlocked, wakes up the first blocked thread and makes it the new lock owner instead of requeuing it. Remaining threads are requeued.

After wake-up, `cond_wait` needs to check the cause of the wake-up: if the thread was requeued, the condition variable must have been signalled, and the caller already owns the mutex. Otherwise, if the timeout expired or the counter's current value mismatched, the caller was not requeued to the mutex' futex and the function needs to lock the mutex again.

Note that the `cond_wait` operation exposes a race condition which may result in a *lost wake-up*. Lost wake-ups are normally prevented by the kernel comparing the futex value, but if – between the time `cond_wait` unlocks the mutex in user space and the time the kernel checks the futex value– exactly 2^{32} wake-up operations are performed, the futex value overflows to exactly the same value and the check would succeed. However, this problem is unlikely to appear in practice, unless the system overloads and low priority waiters do not progress anymore.

Corresponding futex operations in Linux with similar API and behavior are `FUTEX_LOCK_PI` and `FUTEX_UNLOCK_PI` for mutexes, and `FUTEX_WAIT_REQUEUE_PI` and `FUTEX_CMP_REQUEUE_PI` for condition variables [11]. The Linux implementation additionally supports a priority inheritance protocol which is not in the focus of this paper.

C. Futex Operations in the Kernel

We now describe the futex kernel operations. We consider a *wait queue* to be a set of blocked threads waiting on a futex. The kernel creates and destroys wait queues *on demand.* Note that in the following description, a wait queue is *specific to a* *single futex* and is never shared between multiple futexes. The Linux implementation *differs* from this model insofar as the Linux kernel shares a single wait queue for multiple futexes, but the description still matches the Linux model if we ignore unrelated threads in a wait queue and assume that wait queues always exist, as the wait queues in Linux are created at boot time and remain persistent.

Wait Queue Look-up: As mentioned before, the kernel's `futex` operations must relate the user provided futex address to a wait queue by a *look-up function*. If the futex object is shared between processes, the kernel uses the physical address of the futex. For futexes local to the caller's address space, the kernel can use the virtual address for look-up instead. We define further requirements for the corresponding look-up function later. For now, we assume that the kernel maintains *sets* of wait queues and distinguishes local and shared futexes properly, e.g. address space-specific sets for local futexes, and a global set of wait queues for shared futexes.

`futex_lock(&futex, timeout)` handles locking for a mutex. The function first checks whether a wait queue for the futex exists in the set of wait queues. If not, it creates a new wait queue and adds it to the set. Then the kernel evaluates the futex user space value: if the mutex is unlocked, the kernel tries to atomically acquire it for the caller and returns if successful. If the mutex is locked, but the waiters bit is not set, the kernel atomically sets the waiters bit in the futex value. Finally, the kernel enqueues the thread into the wait queue and blocks it with the given timeout. When the timeout expires or the blocking is cancelled for other reasons, e.g. by a signal, the kernel removes the thread from its wait queue. Otherwise, the thread is already successfully dequeued from the wait queue. It is woken up, and becomes the new lock owner. In all error cases, the kernel also removes empty wait queues from the set and destroys them.

`futex_unlock(&futex)` first looks up the wait queue, and if one exists, it wakes up a waiting thread and makes it the new lock owner by updating the futex value in user space. If there are still blocked threads in the wait queue, the kernel additionally sets the waiters bit. Once a wait queue becomes empty after wake-up, the kernel removes and destroys it.

`futex_wait(&futex, compare, timeout, &futex2)` first checks whether a wait queue exists in the set of wait queues, otherwise it creates a new one and inserts it into the set. Then, before enqueuing the calling thread into the wait queue, the kernel checks if the futex user space value still matches the provided compare value, and returns an error if not. The rest of `futex_wait` follows `futex_lock`, but without any updates of the futex value in user space. `futex_wait` accepts an optional second futex which is the target mutex in a requeue operation. `futex_wait` also makes sure that all blocked threads refer to the same second futex (or `NULL`) to simplify the requeue operation.

`futex_wake(&futex, ONE|ALL)` first looks up the wait queue, and, if one exists, wakes up one or all threads. Again, empty wait queues are removed afterwards.

`futex_requeue(&futex, ONE|ALL)` works similarly to `futex_wake`: First, the kernel looks up the wait queue and operates on the given number of blocked threads. Then the kernel requeues threads to their associated mutex wait queue, which it has to look-up as well and possibly create. Eventually, the kernel also checks the mutex value, and if the mutex is currently unlocked, the kernel wakes up the first thread instead of requeuing it, and makes it the new lock owner with the waiters bit set accordingly. The threads are expected to have set a mutex to requeue to, otherwise the call fails.

Locking: All operations also require internal locks in the kernel: Usually, a whole set of wait queues is either protected by a specific lock, or a wait queue provides a specific lock itself (Linux). These internal locks are necessary for the futex protocols to serialize concurrent user space access and concurrent futex operations.

D. Requirements for Determinism

The presented futex operations in the kernel are quite complex. If they are to be used in a real-time system, they must be deterministic, i.e. have a WCET which is (i) *analyzable* and (ii) *bounded*. The main idea is to prevent sharing of wait queues and to use *dedicated* wait queues for each futex instead. This means we have to manage a *set of wait queues* (one for each futex), and each wait queue only contains a *set of blocked threads* specific to the futex. Here we define the requirements for such an implementation:

1) No dynamic memory allocations shall be used for creating wait queues. The problem is simply that dynamic memory allocations can fail at runtime. Also, having fewer dependencies on other components simplifies the WCET analysis.
2) For wake-up and requeuing operations to achieve real-time scheduling, POSIX requires that threads with the highest scheduling priority have to be woken up first. For threads with the same priority, FIFO ordering must be used. This means that wait queues shall be properly ordered.
3) All operations on a set of blocked threads in a specific wait queue i.e. $find$, $insert$, and $remove$ of threads, shall have at worst $\mathcal{O}(\log n)$ execution time, for n threads in the wait queue. This suggests to use *self-balancing binary search trees*, a data structure where the execution time of all operations stays within logarithmic bounds.
4) Similarly, all operations on the set of wait queues, e.g. insertion of a new wait queue into the set, shall have at worst $\mathcal{O}(\log m)$ execution time as well, for m wait queues in the set.
5) `futex_wake` and `futex_requeue` handle a potentially large number of threads in the `ALL` case, so their execution shall be preemptible after handling each thread.
6) `futex_wake/requeue` operations on all threads in a wait queue shall eventually terminate, i.e. threads are not allowed to sneak in into a currently processed wait queue again. This condition follows from the previous requirement that futex operations shall be preemptible.

7) The preemptible operations on all blocked threads in a wait queue shall not be observable by these threads if the threads follow the usage constraints properly. This condition also follows from requirement 5.
8) The implementation should support fine granular locking, i.e. locks on the set of wait queues and a particular wait queue are decoupled to reduce interference between operations on unrelated wait queues.

Note, requirements 3–6 have the same upper bound n, i.e. the overall number of threads in the system, when all threads block on either a different or the same futex. Also, requirements 3 and 4 are *not* required for determinism in the first place, as $\mathcal{O}(n)$ time is deterministic as well. But having an upper bound of $\mathcal{O}(n)$ execution time is only acceptable if n is both known and small. Thus the approach would not be applicable to systems with a very large number of threads.

Preemptible execution of the `ALL` operations is a good compromise with respect to the worst case time an operation holds internal locks, but it introduces its own problems, as requirements 6 and 7 state.

The last requirement helps to simplify WCET analysis, but this is not a hard requirement.

III. Implementation

In this section, we describe our implementation.

As described before, futexes in general require two different data structures in the kernel: (i) a wait queue handling all blocked threads waiting on the same futex, and (ii) a data structure to locate this wait queue, based on the futex user space address as look-up key. We explicitly need this *two-tier design* to isolate threads waiting on unrelated futexes and to support a preemptive implementation of the `ALL` operations.

For both data structures, self-balancing binary search trees (BST) are suitable, e.g. red-black trees or AVL trees. In our futex implementation, we chose to use AVL trees.

Like in Linux and our previous implementation [9], we keep all data related to futex management inside the *thread control block* (TCB) of the blocked threads to get rid of the dependency on dynamic memory management, thus fulfilling requirement 1.

A. Binary Search Trees

From the BST implementation, we require the standard operations $find$, max, $insert$, and $remove$, and additionally $root$ and $swap$. The $root$ operation locates the root node of the BST from any given node, thus requiring that nodes in the BST use three pointers: two for the left and right child nodes, and a third one to the parent node. The $swap$ operation allows to swap a node in the tree with another node outside the tree in $\mathcal{O}(1)$ time without altering the order in the tree. Lastly, the BST implementation requires a *key* to create an ordered tree. The key may not be unique, e.g. threads with the same priority are allowed to exist in the tree. If nodes with duplicate keys need to be inserted, we require FIFO ordering of the duplicate nodes.

B. Wait Queue Look-up in the Address Tree

To locate a wait queue from a futex address, we designate *one* of the blocked threads in a wait queue as *wait queue anchor*. The anchor thread has the root pointer to the wait queue. All wait queue anchors are enqueued in an *address tree*, which is rooted in an *address tree root*.

Key: For shared futexes, we use the *physical address* of the futex as key; and for per-process futexes, we use the *virtual address* as key. Also, both shared and per-process futexes are kept in distinct trees: shared futexes are kept in a global tree shared between all processes, while per-process futexes are kept in process-specific data, e.g. in the process descriptor.

We use the fact that futex variables in user space are 32-bit integers that are aligned on a 4-byte boundary. As the last two bits of a futex address are always zero, we use them to encode further information.

We define that a wait queue is *open* if threads can be added to it, i.e. new threads can block on a futex, and a wait queue is *closed* if new threads can not be added.

We decode the open/closed state of a wait queue in its key: An open wait queue has the lowest bit *set* in the key, for a closed wait queue the bit is *cleared*. By clearing the open bit, we can change a wait queue from open to closed state without altering the structure of the tree. Also, we do not allow open wait queues with duplicate keys, as each key relates to a unique futex in user space. However, multiple wait queues with the same closed key may exist, and they become FIFO ordered due to the ordering constraints in the BST when changing a wait queue from open to closed state. We later exploit this mechanism in `futex_wake` and `futex_requeue` to wake or requeue all threads in a preemptible fashion.

For closed wait queues, we also define a *drain ticket* attribute, a counter value which helps during `ALL` operations later. The drain ticket is a global 64-bit counter incremented each time a wait queue is closed. It should not overflow in practice.

The last specialty in the address tree is the following: if the thread used as wait queue anchor changes, we simply $swap$ the old anchor thread in the tree with a newly designated anchor thread without altering the structure of the tree and we copy the wait queue root pointer, the current drain ticket, and the current open/closed state in the key as well.

This design allows us to perform look-up, insertion, and removal of wait queues in $\mathcal{O}(\log n)$ time, while changing a wait queue from open to closed state and changing the wait queue anchor both need $\mathcal{O}(1)$ time. This fulfills requirement 4.

C. Wait Queue Management

As stated before, the wait queue anchor thread is an arbitrarily chosen blocked thread in the wait queue which holds the root pointer of the wait queue and the open/closed state of the wait queue encoded in the key. We refine this now and define that the thread being the *current root node* of the wait queue is to be used as anchor. If the root node changes due to insertion or removal in the wait queue tree, we swap the root nodes in the address tree as described above. Using the root node thread as its anchor is not mandatory, as any node in the wait queue would do, but this simplifies the implementation when threads are woken up for other reasons, e.g. timeouts, as explained below.

When a thread blocks on a unique futex address, the kernel creates a new wait queue on demand in open state and inserts it into the address tree with this first thread as anchor. Note that this does not involve allocation of memory. Similarly, the wait queue is implicitly destroyed when the last thread (which again must be the anchor) is woken up. The kernel then removes the wait queue from the address tree.

The kernel inserts threads in priority order into an existing wait queue. Also, when waking or requeuing threads, we remove the highest priority thread first.

Removal of an arbitrary node, e.g. on a timeout, requires to find the associated wait queue root to rebalance the tree afterwards. We do not look-up the wait queue in the address tree in this case, as it might have been set to closed state and then up to two look-ups in the address tree would be required. Instead, we simply traverse the wait queue tree to the root node to locate the anchor and remove the thread. This is also necessary when a thread's scheduling priority changes while the thread is blocked. In this case, we remove the thread and re-insert it with its new priority.

If during insertion or removal the wait queue root changes due to the necessary rebalancing in the BST, we transfer the wait queue root pointer and the other current wait queue attributes to the new root and update the address tree accordingly.

This design allows to perform all internal operations on wait queues in at most $\mathcal{O}(\log n)$ time. With it, we are now able to implement `futex_lock`, `futex_unlock`, and `futex_wait` in $\mathcal{O}(\log n)$ time. Also, a `futex_wake` and `futex_requeue` operation targeting a single thread takes $\mathcal{O}(\log n)$ time. This fulfills requirements 2 and 3.

D. Preemptible Operation

We now discuss the preemptibility of `futex_wake` and `futex_requeue` if `ALL` threads need to be handled. In this case, both operations set the wait queue to closed state first, so it can no longer be found by enqueuing operations, then we draw a unique *drain ticket* and save the ticket in the anchor node.

Then the kernel wakes up or requeues one thread after another, but becomes preemptible after handling each thread. After preemption, the kernel is always able to find the wait queue again by looking for the now closed wait queue. If multiple closed wait queues with the same key are found, the drain ticket decides what to do. The FIFO ordering in the BST makes sure that nodes are found with increasing drain ticket numbers. If the drain ticket number of a node is less than the originally drawn ticket, the wait queue relates to an older, but still unfinished operation, and draining older wait queues on behalf of some other thread is fine. So the caller can safely perform its operations as long as the drain ticket number is less than or equal to the drawn drain ticket. The drain ticket is therefore necessary to prevent already handled threads to re-enter these wait queues.

Since at most n-1 threads can be blocked before a draining operation starts and a drain ticket is drawn, the upper limit of steps to complete a `futex_wake` or `futex_requeue` operation is therefore n. This fulfills requirements 5 and 6.

But is it acceptable in general to drain other thread's wait queues? We can answer this question if we look at the following usage constraint of condition variables: the caller of `cond_signal` and `cond_broadcast` shall have the support mutex locked as well, so none of the requeued threads will run before the caller unlocks the support mutex. Therefore, handling threads of a previous waiting round can only happen when `cond_signal` and `cond_broadcast` do not have the support mutex locked, and in this case, POSIX does not longer guarantee "predictable scheduling". This means the answer is yes, and we fulfill requirement 7.

A different use case is a POSIX barrier implementation where a given number of threads block until all threads have reached the barrier. An implementation of `barrier_wait` could then use `futex_wake` to wake all blocked threads. A preemptive `futex_wake` operation could get immediately preempted by a higher priority thread which is woken up as first thread and then the other threads are kept blocked until the original thread continues draining the wait queue. Note that this would not happen in a non-preemptible implementation. However, POSIX also notes that applications using barriers "may be subject to priority inversion" [10]. Alternatively, the barrier implementation can mitigate this issue by temporarily raising the caller's scheduling priority to a priority higher than the priorities of all blocked threads during wake-up.

E. Locking Architecture

The final point to be discussed is the locking architecture to fulfill requirement 8. In this case, we cannot easily provide a solution. We could, for example, implement a nested locking hierarchy where the kernel first locks the address tree, locates a wait queue, locks the wait queue, and then unlocks the address tree again. The strict order in which locks are taken is necessary to prevent deadlocks. But this design approach does not allow to remove an empty (and locked) wait queue from the address tree without holding the address tree lock. Doing this would require unlocking the wait queue first, then locking the address tree, and then finally locking the wait queue again. However, this kind of re-locking exposes races, as the re-locked wait queue may no longer be empty due to concurrent insertion on other processors. And this problem becomes even worse in our design as changes to a wait queue anchor require frequent updates in the address tree.

Still, we assume that a solution can be found, e.g. using a lock-free look-up mechanism in the address tree, but it is still questionable if such an approach would improve the WCET or would simplify the WCET analysis in the end.

For now, we decide to not implement a nested locking scheme as requested by requirement 8, but to use a shared lock for both the address tree and all wait queues. Note that we use dedicated locks for each per-process address tree and the shared global address tree.

Table I
COMPARISON OF FUTEX IMPLEMENTATIONS

	Our new approach	**Our old approach**	**Linux**
Futexes share wait queues	no	no	yes
Wait queue look-up	BST $\mathcal{O}(\log m)$	via `TID` $\mathcal{O}(1)$	hash table $\mathcal{O}(1)$
Wait queue implementation	priority-sorted BST	FIFO-ordered linked list	priority-sorted linked list
- find	$\mathcal{O}(\log n)$	$\mathcal{O}(1)$	$\mathcal{O}(n)$
- insertion	$\mathcal{O}(\log n)$	$\mathcal{O}(1)$	$\mathcal{O}(p)$
- removal	$\mathcal{O}(\log n)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$
Locking	global	global	per hash bucket
`futex_requeue`			
- one thread	yes	yes	yes
- arbitrary number of threads	no	no	yes
- all threads	yes	yes	yes
- preemptive implementation	yes	not needed	no
`futex_wake`			
- one thread	yes	yes	yes
- arbitrary number of threads	no	no	yes
- all threads	yes	not provided	yes
- preemptive implementation	yes	not needed	no
Priority ceiling protocol	yes	yes	yes
Priority inheritance protocol	no	no	yes
	for n threads, m futexes, and p priority levels		

IV. DISCUSSION

In this section, we compare our *new approach* presented in this paper with our *old approach* in [9] and the current Linux implementation in kernel 4.16.

We briefly repeat the key points of our previous implementation in [9]:

- All futex-related data is kept in the TCB.
- Threads on a wait queue are kept in FIFO order.
- Wait queues use linked lists with FIFO ordering.
- For wait queue look-up, the kernel saves the `TID` of the first waiting thread next to the futex value in user space, and updates the `TID` value each time a wait queue changes.
- The *requeue all* operation appends the whole linked list of threads to requeue at the end of the mutex wait queue list in $\mathcal{O}(1)$ time.
- A *wake all* operation is not provided.
- All other operations handle insertion or removal in $\mathcal{O}(1)$ time as well.

Table I shows the differences between the implementations. The complexity of the Linux implementation clearly show that it was designed for the best case, e.g. when only a small number of threads block and collisions in the futex hash table are rare. And this is *usually* the case during normal operation of a system. However, if one considers certification or needs to determine deterministic upper bounds of the WCET, the possible corner cases in the Linux implementation lead to potentially unbounded execution time, e.g. a malicious application could exploit collisions in the hash.

Our old implementation in [9] already addressed these issues, but it does not support priority ordered wait queues which are required for POSIX scheduling. Also, the old implementation does not support POSIX barriers.

Our new implementation presented in this paper is superior in all these respects, however the overhead of a BST compared to linked lists seems quite heavy if the number of used futexes and blocked threads is low. This needs to be evaluated in future work.

Also, our presented locking approach is restricted to a single lock for all futexes, which is worse in the average case compared to the Linux implementation, as Linux uses a dedicated lock for each hash bucket.

Finally, our old and new implementations do not support all futex uses cases available in Linux, as we restrict our implementation to handle either just one or all threads, not an arbitrary number. Regarding other missing features: All discussed approaches can support the priority ceiling protocol defined by POSIX, which adjusts a thread's scheduling priority *before* locking a mutex [10]. But in addition, Linux also supports a priority inheritance protocol for mutexes. This would be possible for our presented design, but this is currently left to future work.

V. CONCLUSION AND OUTLOOK

We have shown an approach to improve the determinism of the kernel parts of a futex implementation by using a two-tier design using two nested self-balancing binary search trees, namely one tree to look up futex wait queues by their address, and a second tree to manage blocked threads in priority order. The shown design has a bounded WCET of $\mathcal{O}(\log n)$ time for all non-preemptible kernel operations with respect to the number of concurrently used futexes and/or blocked threads.

The presented approach is suitable to implement the standard POSIX thread synchronization mechanisms, like mutexes, condition variables, or barriers on top [2]. Also, the presented approach supports the POSIX priority ceiling protocol.

In future work, we would like to improve internal locking in the kernel implementation to reduce interference between unrelated processes. Finally, we would like to evaluate means to support *priority inheritance protocols*.

REFERENCES

[1] H. Franke, R. Russell, and M. Kirkwood, "Fuss, Futexes and Furwocks: Fast Userlevel Locking in Linux," in *Proceedings of the Ottawa Linux Symposium*, 2002, pp. 479–495.

[2] U. Drepper, "Futexes Are Tricky," White Paper, Nov. 2011. [Online]. Available: https://www.akkadia.org/drepper/futex.pdf

[3] "Windows InitializeCriticalSection function." [Online]. Available: https://msdn.microsoft.com/en-us/library/windows/desktop/ms683472(v=vs.85).aspx

[4] B. Schillings, "Be Engineering Insights: Benaphores," *Be Newsletters*, vol. 1, no. 26, May 1996.

[5] D. Hart and D. Guniguntalay, "Requeue-PI: Making Glibc Condvars PI-Aware," in *Eleventh Real-Time Linux Workshop*, 2009, pp. 215–227.

[6] "Windows WaitOnAddress function." [Online]. Available: https://msdn.microsoft.com/en-us/library/windows/desktop/hh706898(v=vs.85).aspx

[7] "Fuchsia zx_futex_wait function." [Online]. Available: https://fuchsia.googlesource.com/zircon/+/master/docs/syscalls/futex_wait.md

[8] "OpenBSD futex manual page." [Online]. Available: https://man.openbsd.org/futex

[9] A. Zuepke, "Deterministic Fast User Space Synchronisation," in *OSPERT Workshop*, 2013.

[10] IEEE, "POSIX.1-2008 / IEEE Std 1003.1-2017 Real-Time API," 2017.

[11] Linux futex manual page. [Online]. Available: http://man7.org/linux/man-pages/man2/futex.2.html