Deterministic Futexes: Addressing WCET and Bounded Interference Concerns

Alexander Zuepke*†, Robert Kaiser*
*RheinMain University of Applied Sciences, Wiesbaden, Germany
†SYSGO AG, Klein-Winternheim, Germany
Email: first.last@hs-rm.de

***Abstract*—Fast User Space Mutexes (Futexes) in Linux are a lightweight mechanism to implement thread synchronization objects like mutexes and condition variables. Since they handle the uncontended case in user space and call the operating system kernel only for suspension and wake-up on contention, futexes are particularly suited for best-effort workloads. Nonetheless, they are widely used to support real-time workloads and are integrated in the Linux real-time patch.**

This paper studies the suitability of the current futex implementation in Linux for certification according to avionics safety standards. Specifically, the paper details the worst-case execution time (WCET) behavior of futexes and the interference patterns that independent applications may encounter when using futexes.

Based on our analysis, we identify some weaknesses in the current futex design in Linux, and we propose and evaluate a futex implementation in the context of PikeOS that is suitable for hard real-time and safety-critical systems which target certification. The proposed solution provides a subset of the functionality of Linux, but reduces WCET complexity and addresses the *Freedom of Interference*-principle for independent applications requested by safety standards.

I. Introduction

In Linux, *Futexes* (Fast User Space Mutexes) are the underlying mechanism to implement POSIX thread synchronization objects like *mutexes* and *condition variables* [1]. Support for futexes was introduced in Linux in 2002 with the Native POSIX Thread Library (NPTL) [2]. One design goal of futexes is to reduce system call overhead for locking objects where possible: futexes do not require any OS support to handle the frequent case of uncontended locking and unlocking operations by using atomic operations, such as *compare-and-swap*. In the contended case, futexes use a generic system call-based mechanism to suspend and wake threads in the kernel. For this, the kernel dynamically creates an internal wait queue based on the futex user space address, and keeps the wait queue as long as threads are suspended. Due to this dynamic handling of wait queues, the futex concept does not enforce restrictions on the number of synchronization objects or on the number of blocked threads, nor does it require prior registration of futex objects in the kernel until contention. Today, futexes represent a versatile compare-and-block mechanism. Based on this mechanism, specific synchronization policies can be implemented in user space. This flexibility makes the *concept* very attractive to operating system developers and has led to its recent adoption by other operating systems, like OpenBSD [3], Microsoft Windows [4], or Google's Fuchsia [5].

In Linux, futexes are also used for synchronization of real-time threads. Both the priority inheritance as well as the priority ceiling protocols are supported, and, in accordance with POSIX real-time scheduling requirements, blocked threads are activated in order of decreasing priority.

Nevertheless, the current Linux futex design does have some shortcomings with respect to supporting hard real-time guarantees. Therefore, its implementation can not be taken as a blueprint/reference for safety-critical systems designed, e.g., for certification according to one of the common safety standards. For example, to look up a wait queue, Linux *hashes* a futex user space address and groups threads with the same hash value into a *shared* wait queue. This can cause interference between unrelated applications accidentally sharing the same wait queue, thus violating the *Freedom of Interference*-principle. Also, a large number of blocked threads in the same hash bucket has an impact on the worst case execution time (WCET) and scalability of futex operations [6], [7]. Additionally, to handle mutexes following a priority inheritance protocol, Linux depends on dynamic memory allocations, but a failed memory allocation may introduce unwanted error conditions which are difficult to handle in user applications. Therefore, it is questionable whether the original design focus on best-effort scenarios makes futexes well suited for safety critical systems.

Safety critical partitioning systems targeting avionics certification according to DO-178C [8] must prevent –or at least limit– any interference between independently executing partitions. This requirement agrees with the guidelines and recommendations provided by the CAST-32A position paper [9], which demands mitigation of interference on multicore systems. Previous recent work such as *Single Core Equivalence* [10] or the *"one-out-of-m" problem* [11] has tackled these interference problems with a co-design of hardware and software partitioning techniques that aim to improve the pessimism in worst-case execution time analysis. This paper addresses these problems at the level of operating system synchronization mechanisms, focusing on futexes.

Specifically, the contributions of this paper are:

- We analyze the Linux futex design.
- We define requirements for *deterministic futexes* with worst case scenarios in mind.
- We provide an improved futex design suitable for hard real-time systems and partitioning kernels with strong isolation guarantees and for targeting avionics certification.

- The futex kernel design prevents any interference when unrelated applications use futexes, and limits the interference when applications share futexes.
- The presented futex design provides only a subset of the features found in Linux, but preserves the look-up of futex wait queues by user space addresses.

This paper extends our previous work on futexes presented at the OSPERT workshop in 2018 [12]. The improved futex implementation is used in PikeOS, a partitioning kernel targeting avionics certification.

The rest of this paper is organized as follows: Section II introduces futexes and discusses a possible futex protocol in detail. Section III discusses the Linux implementation and its drawbacks. We define requirements for a futex design with determinism and reliability in mind in Section IV and present an implementation in Section V. We evaluate the approach in PikeOS and compare it to the current Linux implementation in Section VI. Section VII discusses the results. Section VIII presents related work. We conclude in Section IX.

II. Futex Concepts

A. Terminology and Assumptions

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 may be 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. Upon contention on critical sections in the kernel, *spin locks* are used to serialize threads via busy-waiting on an atomic variable, while *mutexes* use blocking. Lastly, we target systems using *fixed priority scheduling*.

B. Futex General Concepts

A futex is a naturally aligned 32-bit integer variable in user space representing a certain type of synchronization object. The futex value is modified by a type-specific protocol to implement different types of 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 [13].

The kernel uses a futex' user space address as an *index* to a *wait queue* which holds blocked threads referring to the same futex. The main kernel operations on futexes are *waiting* or *blocking* on a futex, *waking up* an arbitrary number of threads waiting on a futex, and *requeuing* a number of threads from one futex' wait queue to another one. To prevent race conditions and *lost wake-up* problems, before actually performing operations on wait queues, the kernel checks if the current futex value still matches the value it had before the system call.

Lastly, futexes come in two flavors: processes can *share* futexes via shared memory segments, while *private* futexes are restricted to a single process. The selected mode affects the kernel's indexing mechanism to locate wait queues.

Table I
Mapping of Futex Operations in User Space and in the Kernel

Futex User Space Operations	Futex Kernel Operations
`mutex_lock`	`futex_lock`
`mutex_unlock`	`futex_unlock`
`cond_wait`	`futex_wait`
`cond_signal`	`futex_requeue` one thread
`cond_broadcast`	`futex_requeue` all threads
`sem_wait`	`futex_wait`
`sem_post`	`futex_wake` one thread
`barrier_wait`	`futex_wait` by earlier threads, `futex_wake` all threads, by last one

C. Futex Operations in User Space

We briefly describe the user space parts of futex operations to help understanding the corresponding kernel parts. Table I shows the mapping of user space operations to kernel ones. As a representative example, we detail the futex-related operations for mutexes and condition variables only. 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 remains the same.

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. Both user space and the kernel understand this protocol. In the fast path (no contention), both `mutex_lock` and `mutex_unlock` try to atomically change the futex value from `0` to `TID` and vice versa, without calling the kernel.

If `mutex_lock` finds an already locked mutex, it atomically sets the `WAITERS` bit in the futex value to indicate contention, then calls the kernel to suspend itself on the futex. 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, if `mutex_unlock` detects that the `WAITERS` bit is set, it calls the kernel to wake a waiting thread. 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 `TID` in the futex value. The kernel also sets the `WAITERS` bit again if other threads are still waiting.

Condition Variable: In condition variables, the futex value represents a counter that is incremented on each wake-up operation. For waiting, `cond_wait` reads the condition variable's counter value, unlocks the associated mutex, and then calls the kernel to block on the condition variable with an optional timeout. Before blocking, `futex_wait` checks if the current counter value still matches the previously read value.

For signaling, both `cond_signal` and `cond_broadcast` increment the counter and call the kernel to requeue either *one* or *all* blocked threads from the condition variable's wait queue to the mutex' wait queue. Before moving threads between wait queues, `futex_requeue` checks whether the associated mutex is currently unlocked. In this case, the kernel wakes up the first blocked thread and makes it the new lock owner instead of requeuing it. All remaining threads are requeued.

After wake-up, `cond_wait` needs to check the cause of the wake-up: if the caller 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 presented `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 erroneously succeed. However, this problem is unlikely to appear in practice.

D. Futex Operations in the Kernel

We now briefly describe the futex operations in the kernel.

Wait Queue: We consider a *wait queue* to be a set of blocked threads waiting on a specific futex. The kernel creates and destroys wait queues *on demand*, i.e. a wait queue exists only during the time threads are waiting.

Wait Queue Look-up: The kernel must relate a futex user space address to a wait queue by an internal *look-up function*. For private futexes, the kernel can use the futex' virtual address for look-up. However, if the futex is shared between processes, the kernel must use the physical address for look-up, as shared memory segments addresses may differ among processes.

Waiting: `futex_lock` handles blocking on a mutex, while `futex_wait` is used for condition variables and other synchronization means. Both functions first check whether a wait queue for the futex exists. If not, they create a new wait queue. Then the functions evaluate the futex user space value: for mutexes, the kernel tries to atomically acquire the mutex if it is unlocked, or sets the `WAITERS` bit if not set. In contrast, `futex_wait` just compares the futex value with the expected value. Finally, both operations enqueue the calling thread into the wait queue and block it with a 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 afterwards. Otherwise, the thread was already successfully dequeued from the wait queue.

Wake-Up: `futex_unlock` and `futex_wake` wake up blocked threads. Both functions first look up the wait queue, and, if one exists, wake up threads. For a mutex, `futex_unlock` wakes up exactly one 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 sets the `WAITERS` bit as well. Similarly, `futex_wake` wakes up a given number of threads, but without modifying the futex value in user space.

Requeue: At first, `futex_requeue` looks up source and destination wait queues, or creates the destination wait queue on demand. If the source wait queue is not empty, the function then requeues a given number of threads. Eventually, the kernel also checks the mutex value, and if the mutex is currently unlocked, the kernel wakes up the first requeued thread and makes it the new lock owner with the `WAITERS` bit set accordingly.

Internal locking: All operations require internal locks in the kernel to serialize concurrent futex operations on wait queues. The kernel also compares the futex value under this lock to prevent *lost wake-up problems*.

III. Futexes in Linux

The analysis of the Linux implementation is based on kernel versions 4.14.59-rt37 and 4.16.12-rt5 with real-time patches.

Based on the kernel's futex abstraction, the C library in user space implements POSIX-conforming synchronization means, i.e. the `pthread` and `sem` APIs [14].

A. Linux Futex API

The original futex API in Linux comprises `FUTEX_WAIT`, `FUTEX_WAKE` and `FUTEX_CMP_REQUEUE` operations [1]. Here, the kernel does not know any details about the futex protocol and therefore does not update the futex value in user space. When mutexes are implemented using this API, then the futex value does not encode the current lock owner [13].

For real-time applications, Linux provides `FUTEX_LOCK_PI`, `FUTEX_TRYLOCK_PI` and `FUTEX_UNLOCK_PI` for mutexes. `FUTEX_WAIT_REQUEUE_PI` and `FUTEX_CMP_REQUEUE_PI` operate on condition variables [15]. The "`PI`" in the function names indicates that the kernel also implements a priority inheritance protocol (`PTHREAD_PRIO_INHERIT`). Except for priority inheritance, the implementation of these functions follows the description in Section II. POSIX additionally defines a priority ceiling protocol (`PTHREAD_PRIO_PROTECT`), but this is implemented in user space by adjusting scheduling priorities.

Threads using the `SCHED_FIFO` real-time scheduling policy are woken up in descending priority order, i.e. highest priority first, with FIFO ordering on priority tie. Threads without real-time requirements are woken up in FIFO order to ensure fairness, i.e. the longest waiting thread is woken up first, but the API does not provide any guarantees about the wake-up order.

Linux supports both private and shared futexes, and uses either a futex' virtual or physical address to look up the wait queues. Additionally, Linux supports robustness: when a thread holding one or more *robust mutexes* dies, the kernel sets the `FUTEX_OWNER_DIED` bit in the futex value and wakes waiting threads. The woken threads detect the dead owner and can try to clean up the stale state, if possible [16].

B. Linux Futex Implementation

Wait Queue Hash and Look-up: The futex implementation in the kernel uses a fixed-sized hash-table of wait queue roots.

The kernel creates this array at boot time and allocates 256 wait queue roots per available processor. The hash table is shared between all processes in the system. To look up a wait queue for a futex, the kernel combines the futex address with internal (process- or file-system-specific) data as *hash key* and permutes the data into a unique index in the hash table. Wait queues are shared between futexes. Due to hash collisions, threads waiting on unrelated futexes may end up in the same wait queue.

Wait Queue: The wait queue implementation is based on a *priority-sorted linked list* (plist), with 140 priority levels (100 real-time priority levels and 40 priority levels for other scheduling classes). The plist implementation comprises a doubly-linked list of the longest waiting threads for each used priority level, and a second doubly-linked list with all threads. Each wait queue root also contains an internal lock to ensure consistency of the wait queue and when checking the expected futex value. For the internal lock, a kernel with the real-time patch uses a priority inheritance mutex. Without the real-time patch, the lock is realized as spin lock.

Waiting: For waiting operations, Linux prepares the futex key based on the futex address and optional timeout data, checks the futex user space value and inserts the thread into the ordered wait queue with the wait queue locked, and finally suspends the thread. After wake-up, the kernel removes the thread under the wait queue lock if the thread is still enqueued on the wait queue and performs necessary cleanups for timeouts.

Wake-up: For wake-up operations, the kernel iterates the wait queue under lock and wakes up the requested number of threads matching the target futex. Threads blocked on unrelated futexes are ignored.

Requeue: For requeuing, Linux locks up to two wait queues in ascending order in the hash. The kernel then iterates the source wait queue, and for threads matching the target futex, it wakes up a specific number of matching threads and requeues any remaining requested number of matching threads to the destination wait queue. Again, unrelated threads are ignored. If requeuing targets a different wait queue, the kernel removes threads from the source plist and adds them to the destination plist, preserving priority and FIFO ordering. Otherwise, threads remain in their current position in the plist and just get their futex key updated.

Priority Inheritance Protocol: The Linux kernel supports a priority inheritance protocol for `PTHREAD_PRIO_INHERIT`. Internally, the kernel maintains *priority inheritance state* (PI-state) data which points to the current futex lock holder and raises its scheduling priority to the maximum priority of all blocked threads. The PI-state data is dynamically allocated on the first call to `FUTEX_LOCK_PI` and freed on the last call to `FUTEX_UNLOCK_PI`. The kernel supports nested PI-mutexes.

Priority Ceiling Protocol: `PTHREAD_PRIO_PROTECT` is an immediate priority ceiling protocol in POSIX. In Linux, this protocol is implemented in the C library by temporarily changing a thread's scheduling priority before locking a mutex and after releasing a mutex. Changing a thread's scheduling priority requires a separate system call.

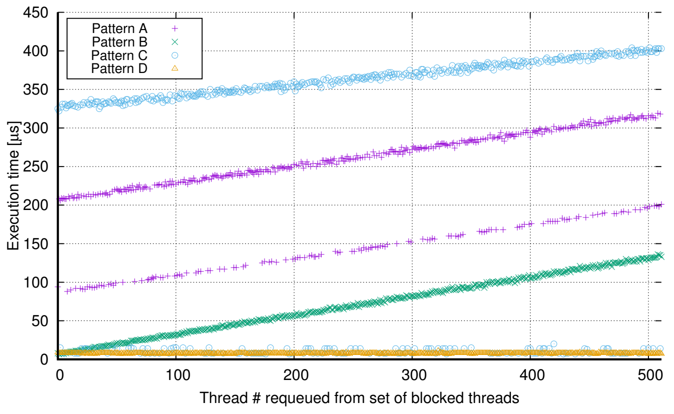

Figure 1. Measured execution times of 512 `FUTEX_REQUEUE(1)` operations on 512 blocked threads on Linux, while another process requeues 512 threads in parallel on another processor. The four interference patterns use different combinations of shared and non-shared wait queues for source and destination futexes in the requeue operation.

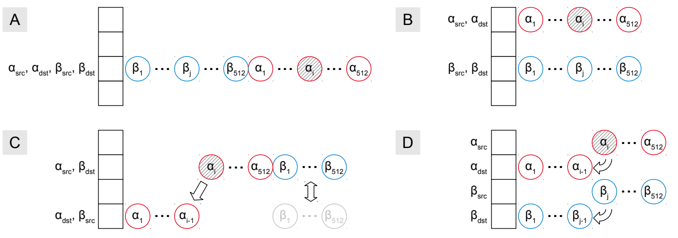
A
$\alpha_{src}, \alpha_{dst}, \beta_{src}, \beta_{dst}$
β_1 ⋯ β_j ⋯ β_{512} α_1 ⋯ α_i ⋯ α_{512}
B
$\alpha_{src}, \alpha_{dst}$
α_1 ⋯ α_i ⋯ α_{512}
β_{src}, β_{dst}
β_1 ⋯ β_j ⋯ β_{512}
C
$\alpha_{src}, \beta_{dst}$
α_i ⋯ α_{512} β_1 ⋯ β_{512}
$\alpha_{dst}, \beta_{src}$
α_1 ⋯ α_{i-1}
β_1 ⋯ β_{512}
D
α_{src}
α_i ⋯ α_{512}
α_{dst}
α_1 ⋯ α_{i-1}
β_{src}
β_j ⋯ β_{512}
β_{dst}
β_1 ⋯ β_{j-1}

Figure 2. Graphical representation of the four interference patterns A-D. Boxes represent four hash buckets and associated wait queues on the right. Circles represent blocked threads in different wait queues while process α requeues thread α_i from wait queue α_{src} comprising threads α_i to α_{512} to wait queue α_{dst} comprising threads α_1 to α_{i-1}. In patterns A and C, processes α and β must lock the same wait queues and block each other. In patterns B and D, α and β can operate on their wait queues in parallel.

C. Real-Time Issues in the Linux Futex Implementation

1) Hash Table with Shared Wait Queues: Futexes of unrelated processes can share the same internal wait queue. Consider an example system where trusted (real-time) and untrusted (non-real-time) applications co-exist. The untrusted applications can interfere with trusted ones by letting a large number of threads block on the same wait queue. In this case, the system integrator must assume worst-case behavior of untrusted applications.

In order to demonstrate this interference, we have devised a simple experiment where two processes α and β on different processors requeue threads. The requeue operation is interesting here because two futex wait queues are involved, a source and a destination wait queue, and both source and destination wait queues can end up in the same hash bucket due to a hash collision. Then, with two processes, we can show different interference patterns of wait queue sharing and of insertion and removal in shared wait queues.

To probe for shared wait queues (hash collisions), the processes both create an array of integers to be used as futexes and monitor the execution time to requeue threads from a source to a destination futex (different indices in the array). The processes build up a list of futexes with hash collisions, as the timing differs minimally when a hash bucket already contains blocked threads. To speed up the tests, we use a patched Linux kernel to retrieve the hash bucket index of futexes and detect collisions directly, as the heuristic approach produces false positives half of the time.

Initially, both processes create 512 threads each and let them block on *private* futexes. Process α (e.g. a real-time application) then requeues one of its 512 threads each time from futex α_{src} to futex α_{dst} (e.g. `cond_signal`) and measures the execution time of each requeue operation. At the same time, process β (e.g. an untrusted application) requeues all 512 threads from futex β_{src} to a second futex β_{dst}, and back.

Both processes execute their operations synchronized: For each measurement, process β starts its execution 0.5 µs before process α, so α effectively should measure the blocking time β introduces. Without interference, a requeue operation of one thread by α usually takes 8 µs, regardless of the number of blocked threads. However, the synchronization in this experiment is not sufficiently precise to catch such short intervals.

Figure 1 shows the measured execution time[1] of each requeue operation of process α with different interference patterns:

- A: all futexes share the same wait queue
- B: α_{src} shares a wait queue with α_{dst}; β_{src} with β_{dst}
- C: α_{src} shares a wait queue with β_{dst}; α_{dst} with β_{src}
- D: all futexes use a dedicated wait queue

Only pattern D shows the expected collision free futex performance (flat line at the bottom). The other patterns show different effects of collisions in the wait queue. Figure 2 shows the status of the wait queues when α requeues a thread α_i in the different collision scenarios. Pattern B shows the *internal interference* α sees when it shares a wait queue for source and destination futexes (no interference from β). As requeued threads remain in their position in the plist, the time to look up a new thread for requeuing in a doubly-linked list takes linearly increasing time. Pattern A shows the effect when all futexes share the same wait queue. Here, the threads of β are kept before α's in the plist. The upper line shows the timing if β starts before α, resulting in the sum of both processes' execution times. The lower line shows the timing if α starts before β (no interference from β), but still the additional 512 queued threads in the plist add a constant overhead. Lastly, pattern C shows *external interference* α gets from β, as β's execution time increases because all its threads always end up behind α's in the source wait queue. Note that outliers –due to the not always reliable synchronization between α and β– have not been removed from Figure 1.

In this experiment, β causes a maximum observed interference of about 400 µs on α, even if both processes do *not* share any futexes. We point out that the scenario in this experiment is highly unrealistic in practice. We deliberately chose this setting in order to demonstrate potential worst-case scenarios, as `FUTEX_REQUEUE` locks two wait queues internally.

2) Resource Exhaustion: A second problem is resource exhaustion in the kernel: Assume a resource-intensive (or a malicious) non-real-time process that allocates all kernel heap. In this case, even if a real-time application, following recommendations for POSIX real-time programming, pre-allocates all its resources, the allocation of the PI-state data structure will fail.

As a consequence, blocking on a mutex with contention fails. The real-time application might be able to revert to a non-PI mutex for blocking, but it cannot apply PI to the lock holder anymore. This might lead to a priority inversion problem in the application. Even if we assume that this problem usually never happens in reality, the Linux futex API specifies an error condition for memory allocation failures. Thus user applications must provide additional error handling code for this. Note that for testing reasons, Linux supports a fault injection interface to let arbitrary memory allocations of PI-state data fail.

3) Arbitrary Priority Boosting: In `FUTEX_LOCK_PI` operations, the kernel can not check whether or not the target thread currently holds a PI-mutex in user space. A malicious thread can now use such an operation to apply a priority boost using the kernel's priority inheritance mechanism to *arbitrary* threads of *unrelated* processes in the system. This not only violates the assumptions of a strict process model where threads can not alter the scheduling of threads in other processes: arbitrarily boosting a real-time thread's priority above others may effectively invalidate a previous scheduling analysis and may lead to priority inversion problems, and boosting of non-real-time threads may lead to starvation of lower priority real-time applications.

However, this problem is not specific to `FUTEX_LOCK_PI`, but to Linux processing in general, as the Linux process model only provides weak separation between processes. As real-time scheduling often requires *root* permissions, a malicious thread could also invoke `sched_setscheduler()` to change the target thread's scheduling attributes directly. The impact of this arbitrary boosting can be mitigated by defining appropriate *control groups* with separate name spaces for processes and thread IDs, but this imposes an additional burden on the system integrator and on the validation of the configuration.

D. Summary

The futex design in Linux strongly focuses on performance, as performance is critical in real-world server and desktop applications. The implementation was carefully tuned to avoid internal overheads where possible. The design is based on the assumption that contention is rare when using mutexes, and, if contention happens, only a low number of threads are blocked at all. This explains the design decision to use a hash table and linked lists, as the necessary operations take $\mathcal{O}(1)$ *amortized* time. Also, Linux developers are aware of the side effects of this design, as [7] and the fault injection interface for PI-mutexes show.

[1]The measurements were performed on an i.MX6 *SABRE Lite* board with four Cortex-A9 ARM cores clocked at 996 MHz and Linux kernel 4.14.59-rt37 with a Yocto 2.4 *Rocko* base system.

IV. Requirements for Determinism

The futex operations in the kernel presented in Section II are quite complex. If they are to be used in a real-time system intended for certification, they must be deterministic, i.e. have a WCET and design which is *analyzable* and *bounded*. Secondly, a futex design should ensure *absence of interference* between independent processes and should show *bounded interference* between processes which share futexes. This allows to separate certification efforts for independent user components and lowers certification costs. In general, a certifiable design requires more scrutiny of corner cases and often sacrifices best-case performance for robustness and determinism.

We now define requirements for a futex design that allows to meet these desirable properties and which indexes wait queues by futex user space addresses, like Linux. See Section VII-E for a discussion on alternative futex designs which index wait queues differently.

Requirement 1. *Each futex shall have a dedicated wait queue.*

We require this to isolate threads waiting on unrelated futexes.

Requirement 2. *Wait queues for private and shared futexes shall be managed separately. Private futexes shall be kept in per-process look-up data structures. Only shared futexes shall use shared data for look-up.*

Separating private futexes removes interference between unrelated processes.

Requirement 3. *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 and certification.

Requirement 4. *Wait queues shall support priority ordering, with FIFO ordering on tie.*

POSIX requires that threads with the highest scheduling priority must be woken up first.

Requirement 5. *All operations on a set of blocked threads in a specific wait queue, i.e. $find$, $insert$, and $remove$ of a single thread, shall have bounded complexity.*

Requirement 6. *All operations on a set of wait queues, i.e. look-up of a wait queue specific to a futex, insertion of a new wait queue into the set, and deletion of an empty wait queue, shall have bounded complexity.*

These two requirements have the same upper bound n, i.e. the overall number of threads in the system, when all threads block on either the same or a different futex. Both requirements suggest to use *self-balancing binary search trees* to bound the complexity to $\mathcal{O}(\log n)$. However, logarithmic complexity is *not* required for determinism in the first place, as $\mathcal{O}(n)$ complexity would be deterministic as well. But $\mathcal{O}(n)$ is only acceptable if n is both known and small. Therefore, the use of logarithmic complexity allows to increase the number of supported futexes and blocked threads. Often, an upper bound for n can be easily approximated, e.g. the amount of available memory limits the number of possible threads. Similarly, logarithmic complexity allows to perform an analysis of the overall system without having detailed knowledge about the applications.

Requirement 7. *Each wait queue should use a dedicated lock, independent of the lock to manage the set of wait queues.*

This requirement requests a fine granular locking scheme to decouple operations on the set of wait queues, e.g. wait queue look-up, from operations on a particular wait queue, e.g. wake-up of a thread. This reduces interference between operations on unrelated wait queues for shared futexes.

However, if this requirement can not be fulfilled, we need to preempt potentially long running operations instead:

Requirement 8. *If locks must be shared between wait queues, wake-up and requeue operations shall be preemptible.*

These operations handle a potentially large number of threads, so their execution must be preemptible, preferably after handling each thread. This is a good compromise with respect to the worst case time an operation holds internal locks.

Requirement 9. *Preemptible wake-up and requeue operations shall eventually terminate.*

A preemptible implementation bears the risk that already processed threads reenter a wait queue again and cause unbounded loops.

Requirement 10. *To affected threads, preemptible wake-up and requeue operations should appear to be atomic.*

Preemption should be transparent to threads on a wait queue and to other threads operating on the same futex, e.g. a preempted futex operation appears to be atomic to them.

V. Implementation

This section describes the futex implementation in PikeOS. The kernel uses two different types of data structures: (i) dedicated per-futex *wait queues*, and (ii) *address trees* to locate the wait queues, using the futex user space addresses as look-up key. This isolates threads waiting on unrelated futexes for Requirements 1 and 2. Like in Linux, all data related to futex management is kept inside the *thread control block* (TCB) of the blocked threads to get rid of the dependency on dynamic memory management, thus fulfilling Requirement 3. Figure 3 shows an overview of the architecture.

A. Binary Search Trees

From the BST implementation, we require the standard operations $find$, min/max, $insert$, and $remove$, and additionally $root$ and $swap$. Nodes in the BST use three pointers: two for the left and right child nodes, and a third one to the parent node. The $root$ operation locates the root node of the BST from any given node in $\mathcal{O}(\log n)$ time. The $swap$ operation allows to swap a node in the tree with another node outside

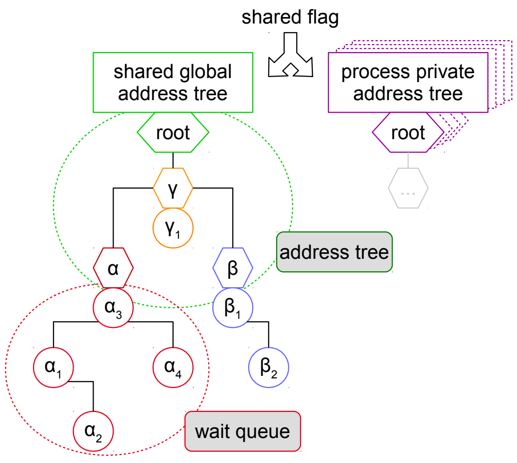

Figure 3. Futex architecture using nested BSTs. The `shared` flag selects between the shared global address tree or a process private address tree. The shared global address tree comprises wait queues (hexagons) α, β, and γ. Address trees are ordered by futex addresses as keys. Wait queue α comprises four waiting threads (circles) α_1 to α_4. Wait queue β comprises two threads β_1 and β_2. Wait queue γ comprises one thread γ_1 only. Wait queues are ordered by thread priority. Threads α_3, β_1, and γ_1 are wait queue anchors of their wait queues. BST node data is kept in the TCBs of the involved threads.

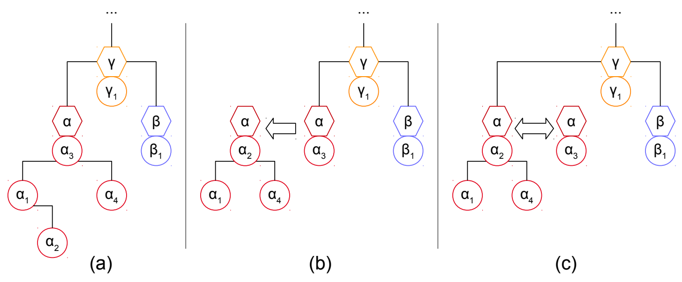

Figure 4. Wake-up of wait queue anchor thread α_3. α_3 is removed from its wait queue, and α_2 becomes the new wait queue anchor. Wait queue state information is copied over to the new anchor in step (b). In step (c), nodes α_2 and α_3 are swapped in the address tree, making α_2 the new wait queue anchor. α_3 is no longer referenced afterwards.

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. We require FIFO ordering of nodes with the same key.

As BST implementation, the PikeOS kernel uses AVL trees. Red-black trees would be suitable as well. However, AVL trees were favored as they are more balanced in the worst case [17].

B. Separate Address Trees for Shared and Private Futexes

Shared and private futexes are kept in different address trees. Like in Linux, the user specifies if futexes are shared or private in a `flags` argument for each futex operation. Shared futexes are kept in a global tree shared among all processes, while private futexes are kept in the process descriptor of each process. This fulfills Requirement 2.

C. Address Tree Management

Address tree: To look up 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 holds 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*. An address tree is ordered by increasing futex addresses as *key*.

Futex key: For shared futexes, the kernel uses the *physical address* of the futex as key; and for private futexes, the kernel uses the *virtual address* as key. We use the fact that futex variables in user space are naturally aligned 32-bit integers. As the last two bits of a futex address are always zero, the kernel uses them to encode further information.

Open/close state: 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. The kernel decodes the 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, the kernel can change a wait queue from open to closed state without altering the structure of the address tree. Also, we do not allow open wait queues with duplicate keys, as each key relates to a unique futex in user space and for Requirement 9. However, multiple wait queues in closed state with the same 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 to wake or requeue threads in a preemptible fashion for Requirement 8.

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

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 needs $\mathcal{O}(1)$ time. This fulfills Requirement 6.

D. Wait Queue Management

Wait queue anchor: As stated before, the wait queue anchor thread is an arbitrarily chosen thread which holds the root pointer of the wait queue and other wait queue attributes. 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 necessary rebalancing in the wait queue tree, the kernel *swaps* the old anchor thread in the address tree with a newly designated anchor thread without altering the structure of the address tree, and then copies over all wait queue attributes, as Figure 4 shows. 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. timeout expiry.

Creation and destruction of wait queues: When a thread blocks on a unique futex address, the kernel creates a new wait queue 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, as the pointers comprising the wait queue are kept in the blocked thread's TCB. 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.

Insertion and removal in wait queues: The kernel inserts threads in priority order into an existing wait queue, with FIFO order on tie. Also, when waking or requeuing threads, the kernel removes 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. The kernel does not look up the wait queue in the address tree in this case, as it might have been set to closed state in the mean time. Instead, the kernel simply traverses the wait queue tree to the root node to locate the wait queue anchor and remove the thread. This is also necessary when a thread's scheduling priority changes while the thread is blocked. In this case, the kernel removes the thread and re-inserts it with its new priority.

This design allows to perform all internal operations on wait queues and the following external operations in at most $\mathcal{O}(\log n)$ time: `futex_lock`, `futex_unlock`, and `futex_wait`. Also, `futex_wake` and `futex_requeue` operations targeting a single thread take $\mathcal{O}(\log n)$ time. This fulfills Requirements 4 and 5.

E. Locking Architecture

In the current implementation, a single shared lock protects both a single (private or shared) address tree and all its associated wait queues. With this, dedicated locks are used for each per-process address tree and the shared global address tree. Although this solution ensures the absence of interference between processes using only private futexes, it still causes bounded contention on futexes shared between processes through the lock of the single shared global address tree. This solution is therefore sub-optimal with respect to Requirement 7.

We note that optimal solutions with respect to this requirement are extremely complex in practice. For example, assume 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 prevents deadlocks, but a potentially empty (and locked) wait queue can not be removed safely 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. This problem becomes even worse in the presented design, as changes to a wait queue anchor require frequent updates in the address tree.

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

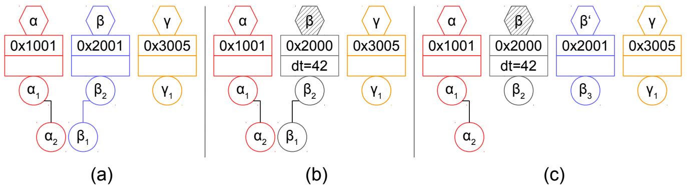

Figure 5. Preemptible draining of wait queue β. Step (a) shows three wait queue α, β, and γ with their futex keys. The lowest bit in a futex key indicates an open wait queue. In step (b), wait queue β is closed and the lowest bit in the key is cleared. The wait queue is assigned a drain ticket of 42. In step (c), while the previous wait queue β is emptied preemptively, a new wait queue β' is inserted with the former open key.

Alternatively, if one relaxes Requirement 3 and allocates a dedicated wait queue object dynamically on demand, an implementation would be possible without the problems listed above. In this case, the wait queue object can be entered into the address tree using the address lock, and then the wait queue object provides a wait queue lock of its own. The only difficulty would be the removal of empty wait queues. Also, such a design would also have a positive impact on the worst-case complexity, as it allows to use a non-preemptible design of wait-queue operations and only the look-up of wait queues in the address tree would cause interference. However, we did not followed this path further, as we wanted to prevent using dynamic memory allocations in the first place.

F. Preemptible Operation

We now discuss preemptible operations on multiple threads: The overview of futex operations in Section II shows that in all cases targeting multiple threads, the kernel always wakes up or requeues *all* threads in a wait queue. Therefore, we restrict the implementation of `futex_wake` and `futex_requeue` to operate on either *one* or *all* threads, but not on an arbitrary number, as Linux allows.

When targeting all threads, both kernel operations set the wait queue to closed state first, then draw a unique *drain ticket* and save the ticket in the anchor node. A closed wait queue can no longer be found by other operations. This prevents already woken or requeued threads from re-entering a wait queue again for Requirement 9, as Figure 5 shows.

Then the kernel wakes up or requeues one thread after another, but provides a preemption point after handling each thread. After preemption, the kernel now looks up the closed wait queue again. If multiple closed wait queues with the same key are found, the stability in the BST makes sure that nodes are ordered by increasing drain ticket numbers. The kernel then continues to perform its operations as long as the drain ticket number is less than or equal to the drawn drain ticket. 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. In this case, the operation drains older wait queues on behalf of other threads as well.

Since at most $n - 1$ threads can be blocked before a draining operation starts and a drain ticket is drawn, the upper

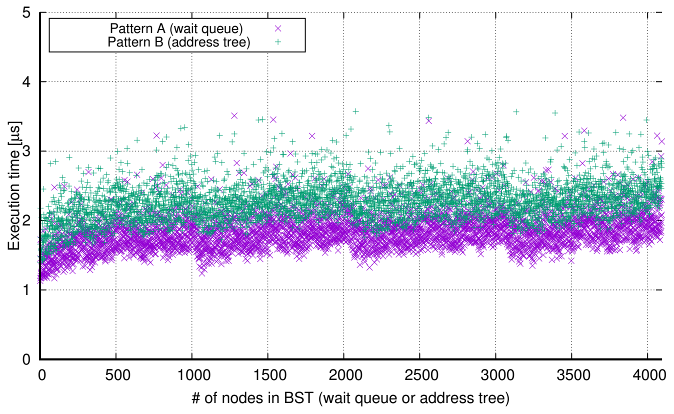

Figure 6. Measured execution times of `FUTEX_WAKE(1)` operations for a variable number of blocked threads on PikeOS. All threads are either waiting on the same futex (test A) or on different futexes (test B). Test A shows the overhead of the BST to manage blocked threads in a wait-queue, while test B shows the overhead of the BST during wait-queue look-up.

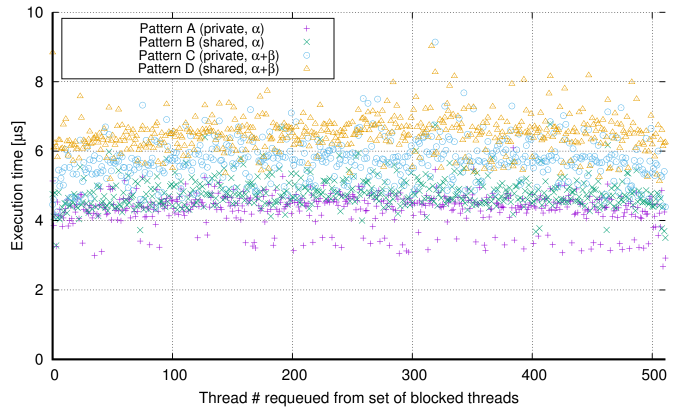

Figure 7. Measured execution times of 512 `FUTEX_REQUEUE(1)` operations on 512 blocked threads on PikeOS. The four tests show different combinations of private (A and C) and shared (B and D) futexes. Additionally, in tests A and B, process α operates alone. In tests C and D, process β requeues 512 threads in parallel on another processor and causes interference.

limit of steps to complete a `futex_wake` or `futex_requeue` operation is therefore n. This fulfills Requirements 8 and 9.

We further discuss Requirement 10 in Section VII.

G. *Priority Inheritance Protocol*

The presented futex implementation does not support a priority inheritance (PI) protocol for mutexes. We discuss this limitation in Section VII.

H. *Priority Ceiling Protocol*

As an alternative to PI, the *Immediate Priority Ceiling Protocol* defined by POSIX is widely used in practice. The presented futex implementation supports it. The protocol raises a thread's scheduling priority to a defined ceiling priority of a mutex before locking the mutex and lowers the scheduling priority after unlocking.

I. *Interference of Shared Futexes*

Another important aspect is to prevent unnecessary sharing of futexes. Like Linux, the presented implementation supports both private and shared futexes. Private futexes are always kept in a process-specific address tree root and therefore do not cause interference with other processes, while shared futexes use a global address tree root and cause interference. To prevent applications from causing interference at all, the use of shared futexes requires an explicit capability in PikeOS.

Therefore, the use of shared futexes requires an analysis of *all* applications using them to determine the WCET of futex operations. Considering a system with k processors and k independent processes using sharing futexes, the worst case interference a process can observe are $k-1$ concurrent futex operations, as kernel operations are preemptive and the kernel uses fair *ticket spin locks* which provide FIFO ordering.

VI. Evaluation

A. *Overhead of Binary Search Trees*

To evaluate the overhead of using two nested BSTs in the presented approach, we conduct another experiment: we measure the execution times of `FUTEX_WAKE(1)` operations for a variable number of blocked threads[2]. We use two different tests:

- A: all threads wait on the same futex
- B: all threads wait on different futexes

Both tests measure the overhead of the BST for the wait queue and the address tree in isolation.

Figure 6 shows the results of tests A and B. Both tests show logarithmically increasing execution times, as the number of nodes in the specific BSTs increases linearly during the test. The spikes at multiples of power-of-two numbers of threads are caused by rebalancing in the AVL tree. Outliers are caused by other system activities, such as interrupt handling.

We can also observe that the timing quickly hits worst case cache behavior, as tree nodes are kept at the same offset within page-aligned TCBs. However, this applies to linked list traversal in the case of Linux as well. The worst cases of the isolated BST operations need to be added to get the overall WCET when both BSTs are fully populated.

In comparison with Linux, where similar operations typically need constant time when there are no collisions in the hash (e.g. pattern D in Figure 1), the BST approach shows a quickly increasing overhead when the number of nodes is small, but the overall overhead is acceptable (min / max / avg in µs: test A: 1.131 / 3.510 / 1.844, test B: 1.400 / 3.574 / 2.273).

B. *Interference of Concurrent Futex Operations*

To evaluate the interference of concurrent futex operations, we conduct the same experiment as for Linux, where process α

[2]The measurements were performed on the same i.MX6 *SABRE Lite* board, but with PikeOS 5.0.2 All futexes are process-private.

benchmarks `FUTEX_REQUEUE(1)` 512 times, while process β requeues 512 threads in parallel on another processor each time. However, as PikeOS does not share wait queues like Linux, we perform additional tests: first, we let process α run in isolation (tests A and B), then concurrently with process β (tests C and D). Also, we measure the difference of private (tests A and C) and shared (tests B and D) futexes.

Figure 7 shows the results. We see that shared futexes are more expensive than private futexes, as the kernel needs to retrieve the physical address of the underlying futex from the page tables. Also, the measured execution time is higher when both process α and β run in parallel. The observed interference in test C with private futexes is additionally caused by *shared hardware*, e.g. the L2 cache and the memory controller.

The actual worst case interference depends on the number of processors concurrently executing operations on shared futexes, and the number of nodes in the BST. The kernel serializes all internal operations on shared futexes with a fair ticket spin lock, which limits this interference to $k-1$ futex operations on a single threads for k processors in a system.

Lastly, Figure 7 shows an arc-shaped execution time for requeue operations which reaches its maximum of $2\mathcal{O}(\log n/2)$ when both source and destination BSTs of a requeue operation are equally populated.

VII. Discussion

In this section, we compare the presented approach with other, related ones, discussing various design aspects.

A. Deterministic Futexes based on Linked Lists

Zuepke presented an implementation of deterministic futexes based on doubly-linked lists [18]. Key points of the design are:

- All futex-related data is kept in the TCB.
- Wait queues use doubly-linked lists with FIFO ordering.
- Insertion and removal of threads takes $\mathcal{O}(1)$ time.
- A `futex_requeue` operation on all threads appends a whole linked list of threads blocked on a condition variable at the end of a mutex wait queue list in $\mathcal{O}(1)$ time.
- A `futex_wake` operation on *all* threads is not provided.
- For wait queue look-up, the kernel saves the thread ID of the first waiting thread next to the futex value in user space, and updates the thread ID value each time a wait queue changes.

Like the futex design in this paper, the linked list approach addresses similar worst case issues and applies similar restrictions to the futex API. However, the approach of wait queue look-up via thread IDs is not free of interference for shared futexes, as all processes need to publicly export their internal thread name space. This suggests to use this approach for use cases with private futexes only, or for a system designs which use a global kernel lock.

Lastly, the linked list approach lacks many features, such as priority ordered wait queues and a futex *wake all* operation, and it supports only mutexes and condition variables.

B. Comparison to Linux

The analysis of the Linux implementation clearly shows 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. This is *usually* the case during normal operation of a system. However, if one considers certification or needs to determine upper bounds of the WCET, the corner cases in the Linux implementation lead to potentially unbounded execution time.

The implementation presented in this paper tries to address these issues, while maintaining a similar feature set. Due to the use of BSTs instead of hashes or linked lists, the presented implementation shows additional overhead with logarithmic complexity in all futex operations, as Figure 6 shows. Also, the locking approach is restricted to a single shared lock, which is worse in the average case compared to the Linux implementation, as Linux uses a dedicated lock for each hash bucket.

The presented design does not support all futex uses cases available in Linux, as it handles either just one or all threads, not an arbitrary number. However, we do not consider this to be a problem, since typical implementations of POSIX synchronization mechanisms do not require operations on an arbitrary number of threads.

With these limitations, it is unlikely that the presented approach would be acceptable for inclusion in Linux. As not all features of the existing API can be provided, the presented approach could be implemented as *additional* futex operations in the Linux futex system call dispatcher, similar to the PI-mutex operations. The effort for this would be comparable to futex implementation in Linux before PI-mutexes, as the complexity of PI handling is not needed.

C. Lack of Support for a Priority Inheritance Protocol

We summarize the rules for a priority inheritance protocol:

- PI is activated when a higher priority thread blocks on an already locked mutex and creates a new wait queue in the kernel.
- The kernel boosts the current lock holder's priority to the maximum priority of all waiting threads.
- The boosted priority needs to be adjusted each time the highest priority thread changes.
- PI is deactivated when a mutex no longer has contention, e.g. the wait queue is destroyed.

The futex implementation presented in this paper does not support PI for mutexes. Still, it would be possible to include the data for PI tracking in the wait queue anchor, e.g. a pointer to the highest priority thread.

However, this would only solve the simple case of *non-nested* priority inheritance. The *nested* case, where a set of threads block on a mutex, but the current lock holder is itself waiting for another mutex, is more complex. Here, the *recursive* nesting creates a *PI-chain*, which can become very long and must not contain cyclic dependencies (deadlocks). At least, implementing fully nested PI with proper deadlock detection would increase the complexity of the `futex_lock` and `futex_unlock` operations to a level not considered acceptable for certification. An implementation with a strictly bounded PI-chain would be possible, and we consider this for future work.

From the kernel implementation's point of view, using a priority ceiling protocol instead of a priority inheritance protocol prevents this complexity, as then any nesting of critical sections must be handled by user space code. However, this also burdens the user space programmer or system integrator to set up ceiling priorities of the mutexes correctly.

Also using a priority inheritance protocol enables further optimization, as various techniques to change a thread's scheduling priority in user space *without* the need for system calls exist [19]–[22]. This allows to prevent any system call overhead for critical sections without contention, similar to uncontended futex operations. PikeOS supports such a technique.

D. Preemptible Operation

Implementing operations on multiple threads in a preemptible way also needs further discussion. In Linux, all operations handling multiple threads execute uninterruptibly w.r.t. other futex operations targeting the same futex, but the presented implementation does not. Preemption in these operations introduces a *sneak-in* problem, where threads can re-enter a wait queue while another thread operates on them. This may facilitate denial-of-service attacks on the kernel, as operations may never terminate. The presented approach with an explicit open/closed state for wait queues solves this, but it introduces the additional problem of multiple wait queues in closed state, which is solved with the drain ticket concept.

The question arises if it is in general acceptable to help out older, but still unfinished operations, i.e. wait queues with a lower drain ticket number. We can answer this question by considering 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 no longer guarantee "predictable scheduling" [14]. This means the answer is yes, and we fulfill requirement 10.

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" [14]. 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. Index-based Futex Design

Alternative futex designs are possible. Compared to Linux and the presented design, the approaches by Spliet *et al.* [19] and by Zuepke *et al.* [21] index wait queues in a table rather than using a wait queue look-up based on futex user space addresses. This yields a simpler design to locate wait queues, and solves problems of internal locking and of interference.

Another alternative would be to use *multiple* address trees and index these address trees via *capabilities*. This would effectively prevent any unwanted sharing of address trees between unrelated processes. However, this adds an additional level of indirection to the futex API. An implementation would then probably skip the look-up of the right address tree and would directly refer to a wait-queue via the capability instead. Assuming a $\mathcal{O}(1)$ capability look-up, such designs can be also considered index-based designs.

However, the size of an index or capability table is usually limited, and index-based futex-based synchronization objects require prior registration in the kernel or a static allocation of index numbers in the application. Also, the different API (index number) does not play well with POSIX synchronization objects placed in shared memory, as indexes and capabilities can differ between processes and only one process' index number can be stored inside a mutex or condition variable object. Therefore, such a design is less generic than the interface originally proposed for Linux, which indexes wait queues implicitly by a futex object's user space address.

We think that both index-based and address-based futex designs have their benefits. One goal of PikeOS is to run uncertified best-effort applications next to certified applications. Best-effort applications lack the rigor needed for static allocation, and therefore, an address-based futex design that meets determinism requirements without neglecting flexibility seems more appropriate. On the other hand, a simpler design like the index-based futex design seems a good fit for systems running only applications that target a high certification level. In most cases, these applications already expose a clear bounded set of synchronization objects, which can be easily mapped to a static index-based design.

VIII. Related Work

To the best of our knowledge, no prior work has analyzed a futex design like in Linux using a wait queue look-up based on futex user space addresses in the context of certification. Zuepke presented an approach for deterministic futexes with FIFO ordering based on doubly-linked lists [18]. Futexes for hard real-time systems using an index-based wait queue look-up were presented by Spliet *et al.* [19] and by Zuepke *et al.* [21].

The observation that kernel overheads are expensive is not new: Liedtke *et al.* discuss this issue for an efficient IPC implementation in the context of L4 microkernels [23]. Wisniewski *et al.* describe *scheduler-conscious* synchronization mechanisms in user space which share information across the application-kernel interface to prevent preemption by the kernel [24]. For synchronization in a Java virtual machine, Bacon *et al.* proposed *Thin Locks*, based on atomic operations for uncontended cases, with a fall-back to OS provided synchronization primitives [25]. Similar approaches where the kernel is entered only on contention are used by *Critical Sections* in Windows [26] and *Benaphores* in BeOS [27].

Futexes extend these prior approaches as a generic *compare-and-block* mechanism. They were first introduced in Linux to implement POSIX thread synchronization objects in user space [1], [2], [13], and then later refined for real-time use cases [15]. Over time, scalability issues were addressed and discussed [6], [7].

Sha *et al.* proposed Priority Inheritance and Priority Ceiling mechanisms to handle priority inversion problems [28]. These mechanisms were later included into operating system standards. To reduce the overhead of frequent scheduling priority changes, Zuepke *et al.* presented an approach to change a thread's priority lazily in user space [20], [21]. Almatary *et al.* present a proof of a lazy PCP protocol in [22]. Spliet *et al.* evaluated the use of other real-time protocols for futexes in LITMUS^{RT} [19].

With SPECK [29], Wang *et al.* present a kernel which provides *scalable predictability*, where predictability bounds observed on a single core, remain constant with an increase in cores. Guiroux *et al.* provide a recent analysis on the performance and scalability of low-level synchronization mechanisms [30].

IX. Conclusion and Outlook

Compared to Linux, we have shown an approach to improve the determinism of the kernel parts of a futex implementation, making it suitable for avionics certification. The futex design uses 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 presented futex design is used in PikeOS and is sufficient to implement the standard POSIX thread synchronization mechanisms, like mutexes, condition variables, or barriers on top. Also, the presented design supports the POSIX priority ceiling protocol.

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 design is interference free when futexes are private to processes, while interference for shared futexes is bounded by the number of processors in the system.

We have reached a point where we have a futex implementation which is bounded, but we have not yet performed a detailed WCET analysis to determine the actual upper bound. This will be part of future work. Also, we would like to evaluate means to support a bounded implementation of a priority inheritance protocol.

Acknowledgment

The authors wish to thank Andrea Bastoni from SYSGO for his input on this paper.

References

[1] H. Franke, R. Russell, and M. Kirkwood, "Fuss, Futexes and Furwocks: Fast Userlevel Locking in Linux," in *Ottawa Linux Symposium*, 2002.

[2] U. Drepper and I. Molnar, "The Native POSIX Thread Library for Linux," Red Hat, Inc, Tech. Rep., Feb. 2003.

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

[4] "Windows WaitOnAddress function." [Online]. Available: https://msdn.microsoft.com/en-us/library/hh706898.aspx

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

[6] D. Bueso and S. Norton, "An Overview of Kernel Lock Improvements," *LinuxCon North America, Chicago, IL*, Aug. 2014. [Online]. Available: http://events.linuxfoundation.org/sites/events/files/slides/linuxcon-2014-locking-final.pdf

[7] N. Brown, "In pursuit of faster futexes," *LWN*, May 2016. [Online]. Available: https://lwn.net/Articles/685769/

[8] "DO-178C: Software Considerations in Airborne Systems and Equipment Certification," Radio Technical Commission for Aeronautics, 2011.

[9] "Position Paper CAST-32A: Multi-core Processors," Certification Authorities Software Team (CAST), Nov. 2016.

[10] L. Sha, M. Caccamo, R. Mancuso, J.-E. Kim, M.-K. Yoon, R. Pellizzoni, H. Yun, R. Kegley, D. Perlman, G. Arundale, and R. Bradford, "Single Core Equivalent Virtual Machines for Hard Real-Time Computing on Multicore Processors," University of Illinois at Urbana-Champaign, Tech. Rep., Nov. 2014. [Online]. Available: http://hdl.handle.net/2142/55672

[11] N. Kim, B. C. Ward, M. Chisholm, C. Y. Fu, J. H. Anderson, and F. D. Smith, "Attacking the One-Out-Of-m Multicore Problem by Combining Hardware Management with Mixed-Criticality Provisioning," in *RTAS*, 2016.

[12] A. Zuepke and R. Kaiser, "Deterministic Futexes Revisited," in *OSPERT Workshop*, 2018.

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

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

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

[16] Futex manual pages. [Online]. Available: http://man7.org/linux/man-pages/man2/futex.2.html

[17] B. Pfaff, "Performance Analysis of BSTs in System Software," in *SIGMETRICS*, 2004.

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

[19] R. Spliet, M. Vanga, B. Brandenburg, and S. Dziadek, "Fast on Average, Predictable in the Worst Case: Exploring Real-Time Futexes in LITMUSRT," in *RTSS*, 2014.

[20] A. Zuepke, M. Bommert, and R. Kaiser, "Fast User Space Priority Switching," in *OSPERT Workshop*, 2014.

[21] A. Zuepke, M. Bommert, and D. Lohmann, "AUTOBEST: A United AUTOSAR-OS and ARINC 653 Kernel," in *RTAS*, 2016.

[22] H. Almatary, N. Audsley, and A. Burns, "Reducing the Implementation Overheads of IPCP and DFP," in *RTSS*, 2015.

[23] J. Liedtke and H. Wenske, "Lazy process switching," in *HOTOS*, 2001.

[24] R. W. Wisniewski, L. I. Kontothanassis, and M. L. Scott, "High Performance Synchronization Algorithms for Multiprogrammed Multiprocessors," in *PPOPP*, 1995.

[25] D. F. Bacon, R. Konuru, C. Murthy, and M. Serrano, "Thin Locks: Featherweight Synchronization for Java," *SIGPLAN Not.*, vol. 33, no. 5, pp. 258–268, May 1998.

[26] "Windows InitializeCriticalSection function." [Online]. Available: https://msdn.microsoft.com/en-us/library/ms683472.aspx

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

[28] L. Sha, R. Rajkumar, and J. P. Lehoczky, "Priority Inheritance Protocols: An Approach to Real-Time Synchronization," *IEEE Trans. Computers*, vol. 39, no. 9, pp. 1175–1185, 1990.

[29] Q. Wang, Y. Ren, M. Scaperoth, and G. Parmer, "Speck: A Kernel for Scalable Predictability," in *RTAS*, 2015.

[30] H. Guiroux, R. Lachaize, and V. Quéma, "Multicore Locks: The Case Is Not Closed Yet," in *USENIX ATC*, 2016.