Deterministic Fast User Space Synchronization

Alexander Züpke
RheinMain University of Applied Sciences, Wiesbaden, Germany
SYSGO AG, Klein-Winternheim, Germany
Email: alexander.zuepke@{hs-rm.de,sysgo.com}

***Abstract*—The Fast User Space Mutex (Futex) mechanism in Linux is a lightweight way to implement thread synchronization objects which handle the uncontended case in user space and rely on the operating system kernel only for suspension and wakeup on lock contention. However, the implementation in Linux today has certain drawbacks that make it unsuitable for use in hard real-time or mixed-criticality systems.**

This paper addresses these issues and presents a novel approach for a futex implementation that guarantees bounded worst case execution time (WCET) in all operations and increases the required level of determinism for safety critical applications. The presented approach of blocking mutexes and condition variables for an unbounded number of threads has linear memory usage and does not require a fine granular in-kernel memory allocator, thus being suitable for real-time operating systems on hardware platforms with low memory or partitioning constraints.

I. Introduction

This paper discusses *fast and lightweight* user space synchronization objects for systems with real-time constraints. We describe a robust and deterministic implementation of *mutexes* and *condition variables*. While mutexes are used to enforce *mutual exclusion* to guarantee exclusive access in resource sharing, condition variables provide a concept for suspension and wakeup-on-notification for producer-consumer problems. Unlike semaphores, which can handle both aspects by the same mechanism, mutexes and condition variables have clearly defined semantics for ordering in the POSIX API [1].

We use a two-tier approach for mutexes and condition variables: The first stage handles the uncontended case and relies on atomic operations in user space. The second stage covers the contended case and uses syscalls (system calls) into the operating system kernel for suspension and wake up. The presented approach is optimized for *best case* usage by omitting expensive syscalls on mutexes with low contention. Similar approaches where the kernel is entered only on contention are used by the *Futex* concept in Linux [2], in Microsoft Windows *Critical Sections* [3], and *Benaphores* in BeOS [4]. As our approach bases upon the Linux concept, we compare both concepts and explain the differences required for determinism.

The novelty and difference of the presented approach in comparison to other existing approaches lies in the way how suspension is handled in the operating system kernel: the kernel does not need to maintain an additional kernel object accompanying the user space object. Removing the kernel object also prevents possible out-of-memory errors on allocation of such objects, or resource shortage by overallocation.

The intended usage scenario covers real-time kernels, space- and time-partitioning, and mixed-criticality environments. A general safety requirement in such systems is, that an abuse of APIs in one partition must not interfere with a correct use other partitions, neither in a temporal sense that the WCET of unrelated partitions becomes indeterminable, nor in a spatial sense that it is possible to disturb operations of unrelated partitions, e.g. by exhausting kernel resources. The intended use case focuses on resource sharing inside a *single* partition utilizing one or more processors. The presented mutexes and condition variables are clearly *not* designed for resource sharing across different partitions or criticality levels in a mixed-criticality system, as they currently cannot handle priority inversion. Both mutexes and condition variables provide *fair* FIFO ordering.

We use the following terminology: a *process* is an instance of a computer program executing in an *address space*. The process comprises one or more *threads*, which are known by the operating system kernel[1] and can be independently scheduled on the processors assigned to the process at the same time. Multiple processes[2] have their own distinct address space each. Processes can share parts of their address spaces with others by using *shared memory*; a shared memory segment is usually *mapped* at different virtual addresses in each address space. A *waiting* thread suspends execution in the scheduler until the thread is *woken up* again on a predefined condition. In this paper, we do not use the terms *job* or *task* which have a different meaning in the field of real-time scheduling analysis. Also, we make no further assumption on scheduling algorithms or thread priorities.

The rest of this paper is organized as follows: Section II explains the futex mechanism in Linux and identifies problems that prevent deterministic use in real-time systems, leading to the requirements described in section III where we present our approach. Section IV discusses mutex and condition variable protocols. We discuss our approach in section V and conclude in VI.

II. Fast User Space Mutexes

A. Linux Implementation

The futex implementation in Linux [2] [5] came with the Native POSIX Thread Library (NPTL) [6] and allows to implement various POSIX-compliant high level synchronization objects such as mutexes, condition variables, semaphores, or readers/writer locks with low overhead in the system's C library in user space. One design goal was to reduce the syscall overhead for these locking objects when possible, thus the implementation uses atomic modifications on user space variables to handle uncontended locking and unlocking solely in user space, and a generic system call-based mechanism to suspend and wake threads in the kernel on lock contention. Basically, a futex is a 32-bit variable in user space, representing a certain type of lock (mutex, condition variable, semaphore) and its value is modified by a type-specific locking protocol.

[1] In contrast to *user-level* threading.

[2] In partitioned environments, a partition consists of one or more processes.

We give a short example of a simple mutex protocol on an integer variable representing the futex: let bit 0 represent the locked state of the mutex, and let bit 1 signal contention. Both bits cleared represents a free mutex. A thread can lock the mutex by atomically changing the lock value from 0x0 to 0x1 using a *Compare-and-Swap (CAS)* or *Load-Linked/Store-Conditional (LL/SC)* operation. A lock operation on an already locked mutex atomically sets bit 1 in the futex to indicate contention, then invokes the `FUTEX_WAIT` syscall to suspend the caller until the lock becomes available again. Symmetrically, when the current lock-holder sees contention during an unlock operation, it clears the locked bit and calls the `FUTEX_WAKE` syscall to wake the first waiting thread that then can acquire the lock itself by atomically setting bit 0 again.

During suspension, the futex syscall allocates an in-kernel futex object and stores the object reference in a hash table indexed by the address of the futex. The futex kernel object comprises the futex address, type, and a queue of waiting threads. The syscall enqueues further suspending threads on the same futex in the existing wait queue that matches the address and type information stored in the futex object. The futex kernel object is freed when the last waiting thread was woken up and the wait queue is empty again.

The third operation on futexes is `FUTEX_CMP_REQUEUE`. It is a special wait queue reassignment method which prevents *thundering herd effects* [2] on signaling of condition variables: instead of waking all threads and letting them compete to lock an associated mutex, the syscall transfers waiting threads from the condition variable's wait queue to the mutex' one. Linux additionally supports *priority inheritance* mutexes and condition variables for threads using POSIX real-time scheduling [7]. Finally, *robust futexes* provide a lightweight mean to notify pending waiters on a crash or deletion of the lock holder.

As futex wait queues are created on demand, Linux imposes no restrictions on the number of user space variables used for futexes. The kernel objects are created on contention only, and therefore the number of kernel objects is limited by the number of threads in the system (when all threads wait on a distinct futex) and by the available kernel memory.

B. Identified Problems and Possible Remedies

The key idea of allowing an unbounded number of futexes in user space and doing kernel operations only on contention appears to be reasonable also for real-time systems. However, the Linux implementation has certain drawbacks that hinder use in deterministic real-time systems:

1) Memory allocations of futex kernel objects require a fine granular in-kernel memory allocator. However, such memory allocations cause *fragmentation*, which is especially bad in space-partitioned environments, where kernel memory resources are limited.
2) Memory allocations can fail due to limited kernel memory. This leads to an extra burden for user applications to handle these kind of failures.
3) User space code controls the addresses of the futex variables which may cause hash collisions.
4) Operations on the wait queue should have deterministic timing for WCET analysis. This is especially important for priority-sorted wait queues. The Linux implementation [8] uses *priority-sorted linked lists* for priority inheritance mutexes and distinguishes 100 priority levels.

In all cases, the number of futex kernel objects is limited by the number of threads that can wait on a futex, which could be all threads in the system. Possible solutions (or reasonable compromises) for these issues depend on the usage scenario: if a fine granular memory allocator is available or out-of-memory failures are acceptable, issues 1 and 2 pose no problem. Alternatively, this could be solved by pre-allocation of the kernel objects, however this pessimistic approach wastes memory for futexes that never see contention. Also, moving the problems of memory allocations to other code sections of an application may not be useful in legacy, non real-time applications. Issues 3 and 4 can also be solved by limiting the number of threads that can utilize the futexes, or by limiting the number of waiting threads at a time, or by limiting the available priority levels. Removing the in-kernel memory allocations promises to solve the first three issues.

III. Deterministic Approach

Here, we present a novel approach without the need for a fine granular in-kernel memory allocator which can handle an unbounded number of threads. Based upon the previous discussion, we first define requirements for a reliable implementation, give an explanation of data structures and operations, and show how to handle wait queues without kernel objects.

A. Requirements

Besides obvious *functional* requirements regarding correct operation of mutexes and condition variables, an implementation must be *robust*, *deterministic* and support *partitioning*:

1) For mutexes, we define the user space operations `mutex_lock` and `mutex_unlock` to acquire and release a mutex lock of type `mutex_t`. For condition variables, `cond_wait` suspends the calling thread waiting on a condition variable of type `cond_t` until another thread wakes either one or all waiting threads by `cond_wake`. While calling a condition variable function, the caller is required to have a *support mutex* locked. This support mutex is released during suspension in `cond_wait`.
2) Support for an unbounded number of these user space objects. The number of objects is limited by the amount of available user memory.
3) No use of an in-kernel memory allocator. All required data in the kernel must be either kept in static *per partition* memory or *per thread* in TCBs (thread control blocks).
4) For robustness, the kernel must correctly handle invalid, unaligned, or unmapped address ranges passed in as references to user space objects.
5) The kernel must not expose loops with CAS or LL/SC operations on user space variables. Otherwise user code could, by updating a variable at the same time, force the kernel into unbounded retries to complete an operation.
6) Bounded operations on wait queues: *enqueue* (append at the end), *dequeue* (remove first) and *requeue* (append one queue to another) operations must be bounded in time, if possible with constant timing characteristics.
7) The kernel must be able to remove a given arbitrary thread from its wait queue in bounded time when handling timeouts, POSIX signals or thread deletion.
8) The lock *scope* defines the grade of required partitioning. A lock with *process local scope* can only be used in its defining process. Further, such locks must not interfere with locks of other unrelated processes. Locks with *global*

scope can be shared between processes[3].

9) For locks with different scope, internal locking of the wait queues must not interfere.

Of the listed requirements, Linux fulfills all objectives except 3 and 9. In the latter case, Linux uses a single hash for all processes. Due to hash collision, a process can delay unrelated ones. Additionally, the priority inheritance versions of Linux mutexes and condition variables violate objectives 5 and 6 by requiring CAS operations in the kernel and using a potentially long running sorting mechanism for the wait queue.

B. Operations and Data Structures

Without an in-kernel memory allocator, the required data to support locking objects in user space must be kept in user space or in the TCB. Wait queues need to be created on demand, so the key idea is to place a token identifying the associated wait queue into user space as well. For syscalls and other functions implemented in the kernel, we use names in capital letters. These operations and data structures are:

1) We define five basic kernel operations that closely resemble their Linux counterparts:
   - `MUTEX_WAIT`: suspend the current thread while waiting to acquire a mutex
   - `MUTEX_WAKE`: wake the next suitable thread from a mutex wait queue
   - `COND_WAIT`: suspend the current thread while waiting on a condition variable
   - `COND_REQUEUE`: move one or all threads waiting on a condition variable to the according mutex wait queue
   - `DEQUEUE`: remove a waiting thread from its mutex or condition variable wait queue

   The first four operations are used by the `mutex_lock`, `mutex_unlock`, `cond_wait`, `cond_wake` operations respectively. The fifth operation (`DEQUEUE`) is used internally by the kernel on timeout expiry, thread deletion, or other waking activities. This limited subset suspends or wakes a single thread at a time. The operations on wait queues are similarly restricted to enqueueing and removal of single threads, or to append complete wait queues to one another.
2) We place the *thread ID* of the head of the queue into user space, next to the futex variable, and a flag indicating if the thread is a legitimate head of a wait queue into TCB. The wait queue itself is kept in kernel space as we keep the internal pointers in the TCB as well. An *indexing* approach with an explicit *ID* identifying the wait queue instead would have required an allocation mechanism again.
3) In the TCB we further store information describing the lock object, i.e. the address in user space, type and scope.
4) We use the scope and the hashed address of the user space object to locate a suitable lock of the wait queue, as described in the following section.
5) For mutexes, we also encode the thread ID of the current lock holder in the futex value in user space.
6) For now, we only provide FIFO ordering of the waiters by using *doubly linked lists*.

By placing data like this, we do not need dynamic allocation of kernel objects. We explain the exact encoding of the current lock holder and wait queue head in section IV.

[3]For locks shared between processes of a single partition, an additional *partition local scope* would be required.

C. Locking of the Wait Queues

The in-kernel wait queues require internal locking during modifications. Skipping the discussion of trivial approaches like disabling interrupts on single processor systems or use of a single global kernel lock on multi processor systems, we must use a scope specific object and/or the address of a mutex or condition variable object as *key* to lock the wait queue.

Depending on the scope of the futex, per-process, per-partition, or global lock objects can be used to protect the wait queues. Per-process and per-partition locks ensure that futex operations of concurrently executed processes do not interfere with each other. For mixed-criticality systems, access to locks of higher critical partitions should be privileged.

Using the futex address as key, multiple locks could be used. The virtual address is sufficient to uniquely identify a futex in a single process.[4] For locking scopes beyond a single process, the physical address must be used. Eventually, hashing of the address leads to the right lock in a scope specific array.

We assume that these locks are located in scope specific data storage, e.g. process descriptor or global data, and exist during the lifetime of the scope. The actual number of hashed locks is a trade-off between scalability and memory usage. The choice depends on the targeted system environment.

IV. Locking Protocols

We now describe the necessary data structures and state transition protocols of a futex variable representing a mutex and a condition variable. We omit all error checking except for the sequences where we must retry an operation. APIs are reduced to the minimum necessary to describe the concept.

We assume the following scenario: Firstly, the target architecture has 32-bit atomic CAS or LL/SC instructions. Secondly, all threads can be referred by unique IDs of *less* than 32-bit, with the value zero denoting an invalid thread ID. Lastly, we limit the scope to a single process only, so that virtual addresses are used for locking of the wait queues[5].

A. Mutex Protocol

A mutex in user space comprises elements $\langle T, W, Q\rangle$ in two consecutive 32-bit integers, see figure 1. Let the *lock state* S be the first integer encoding T and W: The *waiters bit* W is a single bit indicating whether or not the wait queue is empty. The *lock holder's* thread ID T is encoded in the remaining bits. The second integer points to the *wait queue head* and holds the thread ID Q of the oldest thread in the wait queue or zero if the wait queue is empty, see step c in figure 2.

We use the notation $S = \langle W, T\rangle$ to describe the state of the mutex, and $Q = \{...\}$ to describe the wait queue as FIFO-ordered set, pointed to by the thread ID of the left-most thread on the queue. The mutex is *free without waiters* if both the values of S and Q are zero: $S = \langle 0, 0\rangle \wedge Q = \emptyset$, which is also the initial state of a mutex. Further, we let l, l' and l'' describe threads attempting to acquire the lock or suspended waiting on the lock. When used in conjunction with S or Q, l, l' and l'' describe the respective thread IDs. The rules and invariants to access S, Q, and the internal wait queue are:

1) The user changes only S using CAS.
2) The kernel accesses S, Q, and the wait queue only while holding the according wait queue lock.

[4]We neglect the case of mappings of the same physical memory to different virtual addresses. This will be detected and result in an address mismatch error.

[5]For shared locks, the lock object's physical address should be used.

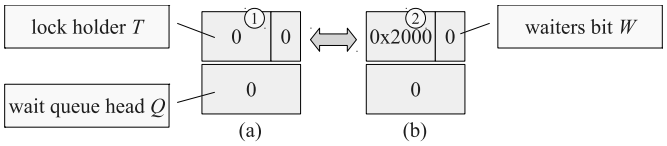

Fig. 1: Example of mutex state changes in the uncontended case: `mutex_lock` and `mutex_unlock` change T atomically, 0 denotes a free mutex and $0x2000$ refers to a lock holder's thread ID.

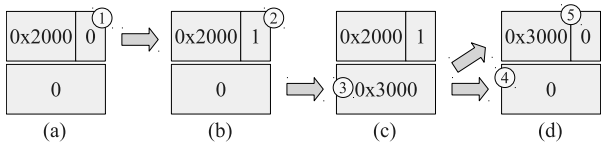

Fig. 2: Example of mutex state changes in the contended case: on contention, `mutex_lock` sets W in step b and calls the kernel. `MUTEX_WAIT` enqueues the calling thread $0x3000$ and updates Q in step c. A later `mutex_unlock` by thread $0x2000$ sees contention and let `MUTEX_WAKE` put in the first waiter as new lock holder in step d. As $0x3000$ was the only waiter, the wait queue becomes empty, Q and W are set to zero.

3) The user atomically sets T in S the first time on contention.
4) The kernel clears T if the wait queue is empty and there is no more contention.
5) If Q is zero, the wait queue is empty.
6) If Q is non-zero, it points to the head of the wait queue.
7) If the head of the wait queue changes, the kernel updates Q to the new head's thread ID.
8) If the wait queue becomes empty, the kernel sets Q to zero.
9) On suspension, the kernel adds the waiting thread to the wait queue (or creates a new one if it was empty before).
10) On wake up, the kernel removes the first thread from the wait queue.

In detail, the operations for mutexes are:

1) On an uncontended lock operation by thread l, `mutex_lock` atomically changes S from $\langle 0, 0\rangle$ to $\langle l, 0\rangle$. An uncontended unlock operation reverses this, see figure 1. The content of Q is ignored.
2) On a contended lock operation (non-zero S) by l', `mutex_lock` atomically sets the W bit in S if it is not yet set ($S = \langle l, 1\rangle$) before suspending in the kernel using `MUTEX_WAIT`, see step b in figure 2. It further passes the current value of S to the kernel as S'.
3) `MUTEX_WAIT`: After locking the wait queue of the mutex, the call checks if S' equals S in user space. This prevents *lost wakeup* errors, because the update of S and the syscall `MUTEX_WAIT` are not atomic.

   Before enqueueing l', the kernel reads Q first:
   - If Q is zero, the wait queue is empty and the kernel sets Q to l', creating a new wait queue, see figure 2, step c.
   - If Q is non-zero, the kernel tries to find the referenced thread and compares the mutex attributes to check if the referenced thread is an eligible wait queue head and waiting on the same mutex. It then inserts l' into the existing wait queue.

   Finally, the kernel releases the wait queue lock and suspends the calling thread in the scheduler.
4) On a contended (W set) unlock operation by l, `mutex_unlock`() relies on `MUTEX_WAKE` to wake the next waiting thread and pass the lock over to it.
5) `MUTEX_WAKE` locks the wait queue first, then checks Q:
   - If Q is zero, the wait queue is empty. Then it sets S to zero and the operation completes. This describes the race that leads to the lost wakeup error.
   - If Q is not zero, the call checks the wait queue; the denoted thread must be an eligible wait queue head and point to the same mutex.

   The kernel then removes the first thread from the queue to become the next lock holder. The call updates Q with the ID of the next thread in the wait queue, or zero if the queue is empty. Then it sets T to the ID of the new lock holder, with W set accordingly, see figure 2, step d. Finally, it unlocks the wait queue and wakes up the new lock holder.

It is necessary to let the kernel update S and Q consistently without using CAS, otherwise user code could force the kernel into endless CAS retries. The parts in user space retry their operation if either the CAS fails or the comparison of S' and S in the kernel fails. The shown implementation is a basic version only. A non-suspending `mutex_trylock` operation can be deduced from the first step. It is possible to busy-wait in `mutex_lock` for a certain time before suspending in the kernel or to add timeouts to the call.

B. Condition Variable Protocol

The condition variable protocol is similar to the mutex protocol. A condition variable comprises elements $\langle C, Q'\rangle$ in two consecutive integer variables. The *condition counter* C in the first integer is a single counter incremented on every wakeup operation. The second integer Q' describes a *wait queue head* again and identifies the longest waiting thread in the queue. To describe the mutex associated with the condition variable, we use $\langle S, Q\rangle$ again. The initial state of a condition variable is $C = 0 \wedge Q' = \emptyset$. For C and Q', the same rules apply as described for S and Q. The operations on condition variables are:

1) `cond_wait` is used to wait on the condition variable. The call reads C and provides it to `COND_WAIT` as C' to detect lost wakeup errors. Before calling the kernel, `cond_wait` releases the accompanying mutex.
2) `COND_WAIT` performs similar steps than `MUTEX_WAIT`: it locks the wait queue, compares C' with C in user space, enqueues the calling thread in the wait queue, and suspends the calling thread. Additionally, the call keeps a reference to the mutex for later `COND_REQUEUE`.
3) `cond_wake` wakes one or all threads. It first increments C and then calls `COND_REQUEUE`. As the caller is required to hold the mutex while calling `cond_wake`, the update of C is atomic to others.
4) `COND_REQUEUE` locks the wait queue of the condition variable and checks Q' as shown in figure 3:
   - If Q' is zero, the call bails out, as no threads are waiting.
   - Otherwise it removes either one or all threads from the wait queue (step 3). When all threads are to be migrated, the syscall keeps the complete queue as is and appends it at the end of the mutex wait queue later.

   After updating Q' accordingly, `COND_REQUEUE` unlocks the first wait queue. Then, by the reference to the mutex that was kept in the TCB of the waiters, the syscall locks the mutex wait queue (step 4), checks, appends previously

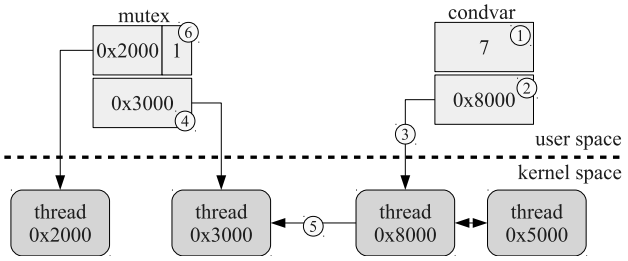

Fig. 3: Example of a COND_REQUEUE migrating all threads: The call disconnects the condition variable's wait queue (starting with thread $0x8000$) and sets Q' to zero. Then it attaches the wait queue to the one of the mutex with thread $0x3000$ as head.

removed waiters (step 5), and updates Q in user space if necessary. Additionally, the waiters bit S is set if necessary before unlocking.

It is also possible to design COND_REQUEUE in such a way that the caller is not required to hold the mutex. Then the first waiter is made the lock holder of the mutex S if it is currently free. However, this would require a CAS operation on S which could be forced into unbounded retries by user space code.

The wait operation exposes a race condition between the time it unlocks the mutex in user space and the time the kernel checks C. Lost wakeup issues are normally handled by the kernel comparing C and C', but if during that time exactly 2^{32} wakeup operations are performed, C overflows to exactly the same value and the check succeeds. Developers must be aware of this *ABA problem* [9].

C. Dequeue Operation

A generic *remove from wait queue* operation is required when waiting must be interrupted, for example when a timeout expires or the thread is going to be deleted. On DEQUEUE, the kernel does not know if the thread is currently suspended on a condition variable or mutex wait queue. Therefore it locks both wait queues at the same time. This can be done safely when both user space objects refer to different addresses and locks are taken in order of ascending addresses. After removing the thread from the lock, DEQUEUE checks if the thread was referenced in Q or Q' and updates Q or Q' accordingly. If Q becomes zero, the call clears the waiters bit in S as well.

V. Discussion

Here we discuss robustness and determinism aspects of our approach. Basically, we have to consider that user code may accidentally or deliberately manipulate either S, C, Q or Q'.

Until now, the encoding of the current lock holder in S is just of informative nature and has no impact on the operation of the kernel. More elaborate versions of the mutex user space implementation may implement deadlock detection or a recursion counter and may depend on S being correct, but the problem is solely a problem of user space. The same applies to C. The values of Q and Q' are of more importance to the kernel. If these are changed, the kernel cannot find the wait queues any more. However, the wait queue itself is kept in kernel space and therefore its integrity is not affected.

The kernel is robust against the following errors:

1) Q or Q' are set to zero. Threads can no longer be woken up on calls to mutex_unlock or cond_wake. But the kernel always allows to cancel the waiting operation by other means using DEQUEUE.
2) Q or Q' are set to another thread currently waiting in the queue. The kernel detects this by checking if the thread is an eligible wait queue head. For FIFO queueing, the check could be omitted, which then just affects the order in which threads are woken up. Otherwise it would take an unacceptable time of $O(n)$ to locate the original head.
3) Q or Q' are set to threads currently waiting in different wait queues, or threads not in waiting state, or invalid thread IDs. Again the kernel detects such conditions.
4) Q or Q' are set to threads out of the scope of the lock. The kernel can detect such errors by other means (e.g. communication capabilities, not shown).

We can consider that all these errors have the same impact, namely than an application may not release a locked mutex. The fault remains isolated in the application and can not harm the kernel or violate partitioning.

From a temporal point of view, none of the operations requires searching or exposes unbounded behavior. All loops are bounded to $O(1)$ for WCET analysis.

VI. Conclusion and Outlook

We have shown a novel approach to implement an unbounded number of mutexes and condition variables using a two-tier approach, with user space handling the uncontended case and the kernel handling contention, and described the low level protocols to be followed. We have also shown how to implement futexes without a fine granular memory allocator.

The presented approach is a foundation to implement other synchronizations means like semaphores, reader-writer locks or barriers on top [5], or to extend the implementations with a safety net of additional error checking or convenience functionality like recursive mutexes. It is also possible to implement timeout handling in all waiting operations or busy-wait on a mutex for a certain time before suspending in the kernel.

In future work, we would like to add support for priority ordered wait queues in fixed-priority scheduling use cases. Also we like to discuss the practicability of the approach when using dynamic priority scheduling like EDF. Finally, we would like to evaluate means to prevent *priority inversion*. At least, a *priority inheritance protocol* seems implementable.

References

[1] IEEE, "POSIX.1-2008 / IEEE Std 1003.1-2008 real-time API," 2008.

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

[3] "Initializecriticalsection function." [Online]. Available: http://msdn.microsoft.com/en-us/library/windows/desktop/ms683472%28v=vs.85%29.aspx

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

[5] U. Drepper, "Futexes are tricky," White Paper, Nov. 2011. [Online]. Available: http://people.redhat.com/drepper/futex.pdf

[6] U. Drepper and I. Molnar, "The native posix thread library for linux," Red Hat, Inc, Tech. Rep., Feb. 2003.

[7] D. Hart and D. Guniguntalay, "Requeue-pi: Making glibc condvars pi-aware," in *Eleventh Real-Time Linux Workshop*, 2009, pp. 215–227.

[8] "Lightweight pi-futexes." [Online]. Available: http://lxr.linux.no/#linux+v3.8.7/Documentation/pi-futex.txt

[9] M. M. Michael, "ABA prevention using single-word instructions," IBM Thomas J. Watson Research Center, Tech. Rep. RC23089, Jan. 2004.