I would hate user space locking if it weren't that sexy...

(fusyn+RTNPTL: The making of a real-time synchronization infrastructure for Linux)

`http://developer.osdl.org/dev/robustmutexes/`

Iñaky Pérez-González
inaky.perez-gonzalez@intel.com

Boris Hu
boris.hu@intel.com

Salwan Searty
salwan.searty@intel.com

Adam Li
adam.li@intel.com

David P. Howell
david.p.howell@intel.com

Linux OS & Technology Team, Intel Corporation

Abstract

Linux has seen a lot of new features and developments in the last years in order to accommodate better scalability, interactivity, response time and POSIX compliance. With these changes, Telecom developers began to get serious about using Linux and started porting their systems to it. By doing that they brought new usage models and needs to the community; and among those needs was support for threads, mutual exclusion, priority inversion protection and robust synchronization for mission critical and fault-proof systems on both timesharing and soft real-time environments. This paper describes our experiences trying to meet this need, the current state and where are we headed. We will detail how originally we tried to modify the futex code, but later found we had to abandon that in favor of a similar design based on a layered implementation. This implementation accommodates a kernel and user space locking and synchronization infrastructure that will meet the requirements of those applications needing to use and port complex multithreaded real-time code.

1 A look at the requirements

The Carrier Grade Working group, or CGL, was created under the auspices of the OSDL; it provides a meeting point for all parties who share an interest on Linux use for Telecom: network equipment vendors, Linux distributors and developers, carriers, etc.

It was in this forum where missing features were identified. Carrier Grade Linux needed good soft real-time[1] features, specially with multi-threaded programs. As well, it needed a common feature provided by Solaris' mutexes

[1]For short, we'll use real-time to refer to *soft* real-time.

that was not present in Linux: *robustness*.

This project was started to provide a kernel synchronization infrastructure (*fusyn*) with the indicated characteristics, as well as the proper modifications to the NPTL user space library (*RTNPTL*) for it to use the new infrastructure and provide the new features.

The basic immediate requirements could be summarized in:

- The infrastructure should provide the primitives needed by NPTL to support the following POSIX tags:
  - TPS: thread priority scheduling
  - TPI: priority inheritance in mutexes
  - TPP: priority protection in mutexes

  Or simply: *anything that is needed for soft-real time support*.
- The implementation should support robust mutexes similar to those of Solaris.
- The implementation should provide equivalent features at the kernel level for use by drivers and subsystems.

With this in mind, we aimed to satisfy the following detailed requirements:

1. mutexes and conditional variables must work according to real-time expectancies
   (a) All operations (lock, unlock, priority promotion and demotion, etc.) should be deterministic in time, and *O(1)* when possible (except of course, for waits).
   (b) The order of lock acquisition by waiters (in mutexes) and wake up (in conditional variables) has to be determined by the scheduling properties of each blocked task/thread.
   (c) Minimization of priority inversion (given the importance of this item, it will be treated in its own section):
      i. lock stealing: in SMP systems, during on the acquisition of the lock a lower priority thread can steal the lock from a higher priority thread.
      ii. when a high priority thread A is waiting for a lower priority owner B to relinquish the mutex and B is preempted by a medium priority thread C.
         A. priority protection
         B. priority inheritance
2. Robustness: when a mutex owner dies, the mutex switches to a *dead-owner* state and the first waiter gets ownership with a special error code.
3. Uncontested locks/unlocks must happen without kernel intervention.
4. Deadlock detection

As well, in order to provide the benefits of this infrastructure to all the levels of a Linux system, it must be possible to use it not only by the user space code, but also by the kernel code.

1.1 The real time expectancies

Real-time is all about being **deterministic**, so all algorithm execution times need to be as predictable or bounded as possible. Using $O(1)$ algorithms helps with this [2].

[2] It is possible to be deterministic with a $O(f(N))$ operation, as long as $f(N)$ is known; however, in most, if not all, of the cases involving mutex operation, it is highly impractical or plainly impossible to find out $f(N)$, and thus a possibly simpler implementation has to be replaced with one potentially more complex, but $O(1)$.

POSIX dictates that upon unlock of a mutex, the scheduling policy shall determine who is the next owner. An obvious way of doing this would be to wake up all of the waiters and let them compete for the lock–the scheduler would determine that the highest priority task would get there first.

However, this causes scheduling storms, unnecessary context switches and general avoidable overhead. It is easier and more effective to determine which is the highest priority waiter and only wake that one up. To implement this task in an $O(1)$ way, we need to queue the waiters in a sorted list that provides constant time queuing and unqueuing. On unlock or wake up time, the first waiter in the list will be the highest priority one.

1.2 Priority inversion

This condition happens when a lower priority thread blocks a higher priority one. The most general case (Figure 1) is the lower priority thread that holds a resource needed by the higher priority one–a situation that has to be avoided–as much as possible. As indicated before, we aim to solve three different flavors.

The first is **lock stealing**. For performance reasons, to avoid the convoy phenomenon[3] [1], the *unlock* operation is done by unlocking the mutex and then waking up the first waiter (eg: A). The waiter claims the mutex and then becomes owner. On a single CPU system it can be preempted only by higher priority tasks[4], so lock stealing is not a problem; however, on multi-CPU systems, a lower priority task C running on another CPU could claim the lock just before B had the chance to do it and it would create a priority inversion scenario (see Figure 2).

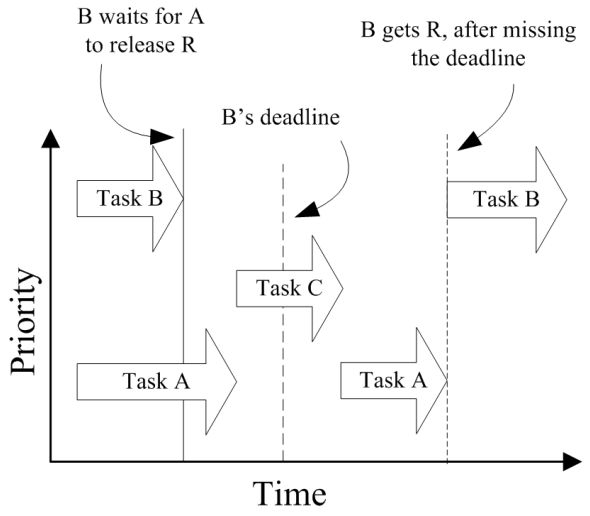

Figure 1: A case of priority inversion: high-priority task B misses its deadline because lower-priority task A holds for too long a resource it needs, as mid-priority task C preempted it. A lower priority task C blocks a high-priority task B.

The solution to this problem is simple: do not unlock the mutex, just transfer the ownership without unlocking it. We call this *serialized* unlock (versus *parallel*, wake and claim). This method severely limits performance in many cases, because it forces a context switch (causing the already mentioned convoy phenomenon). There has to be a compromise between protection and performance and by offering the option to unlock a mutex in either way, a developer can dynamically adapt according to her needs.

The other two cases (of priority inversion) are more complex. They solve the scenario depicted in Figure 1 where task B is waiting for a mutex owned by task A and task C preempts task A. When the priorities are $p(B) > p(C) > p(A)$, we have a priority inversion; task B will miss its deadline because C is blocking A from

[3] Summarizing: if task A (high priority) unlocks by transferring ownership to the first waiter B (lower priority), it forces a context switch to B, and if then A recontends for the lock it will create a convoy of waiters that is difficult to dissolve.

[4] We will use the terms tasks or threads indistinctly to refer to any entity that can acquire a mutex.

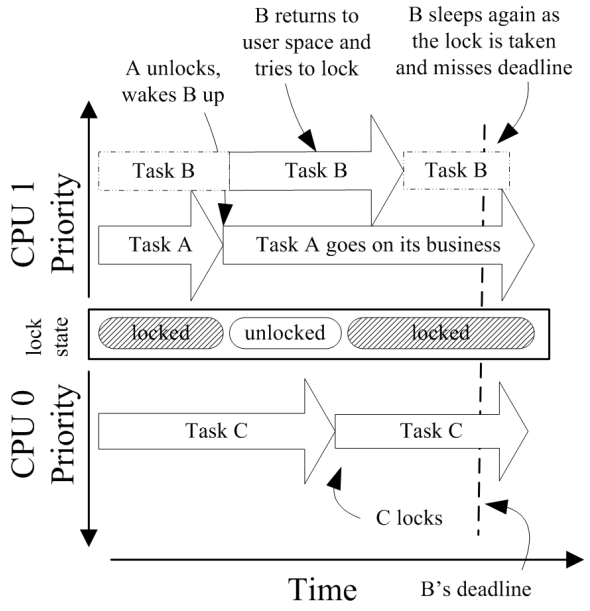

Figure 2: Low priority task C running on CPU0 steals the lock from higher priority task B running on CPU1.

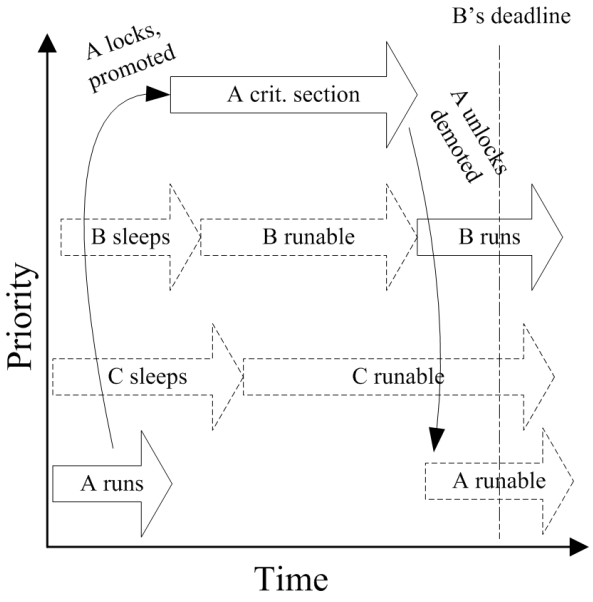

Figure 3: Priority protection: task A locks and its priority is promoted to the prioceiling; task C cannot preempt it and it finishes its critical section (and is demoted) in time for B to meet its deadline.

completing its mutex-protected critical section.

There are different ways to deal with this problem, but the most common involve bumping up the priority of the owner of the lock to a certain value.

In **priority protection** (or PP), a *priority ceiling* is determined as part of the design cycle. This is normally the highest of all the priorities among the threads that will share a given mutex; as soon as it is locked, the priority of the owner is raised to match that of the priority ceiling (see Figure 3. When a thread owns many priority-protected mutexes, its priority is that of the highest ceilings. This approach is simple and guaranteed to be trouble free. However, it is laborious; determining the priority ceiling might not be an easy task at all in a moderately complex system where modules from different parties need to interact.

Enter **priority inheritance** (PI): in this case we have a similar situation, but there is no priority ceiling. What happens in this case is that the priority of the owner is boosted up to that of the highest priority waiter, the first one (see Figure 4). Similarly to the previous case, if a task owns many PI-mutexes, its priority will be the highest of them all. There is no need now to do design-time analysis; the system solves it automatically. Of course, there are drawbacks– it does not come for free. This operation is more expensive, especially in the presence of owner/wait chains[5]. The propagation of the priority boost can be long (and will be $O(N)$ on the depth of the chain) and this can lead to unexpected surprises if the interaction across different threads and mutexes in the system is not kept on a tight leash (see [2] and [3]).

[5]Task A waits for mutex M that is owned by task B that is waiting for mutex N that is owned by task C that is waiting for mutex O…

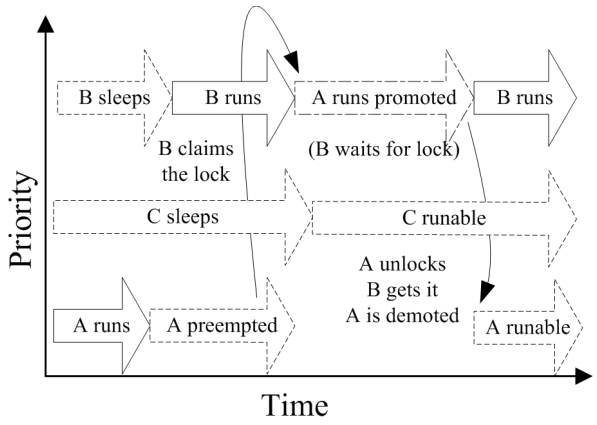

Figure 4: Priority inheritance: Task A (lock owner) is promoted to task B's priority when B waits for the lock; as soon as A unlocks, it gets demoted and B gets the lock. C never has a chance to preempt A.

Priority inheritance needs to be used with care– it is not a straight solution for a system with deadlock problems to make a mutex PI. What if that mutex is being shared with some low priority timesharing task that is not aware of the fact? In these cases, if a task does some kind of CPU spinning, the system is dead. The concept of priority inheritance and the simplicity it gives to designs provides enough rope as to hang oneself, as the effects can propagate way far more than expected.

1.3 Robustness

Mutex robustness is a key feature for implementing systems tolerant to certain kind of failures. A certain task A is holding a normal, *non-robust* mutex M with one or more waiters W_n blocked in the kernel. If it receives a fatal signal and is killed, the mutex will still be locked and the waiters will be never woken up. There are different ways to detect and recover from this situation, but they usually involve painful and complicated designs with watchdogs, timeouts, etc.

Robustness embeds all these in the mutex mechanism. When a task owns a mutex, the mutex knows who is its owner, and asks to be notified if the owner dies. If and when this happens, the mutex will be moved to an special *consistency state*, *dead-owner* and effectively unlocked; this will give control to the first waiter (or remain unlocked as *dead-owner* until somebody else claims it).

Threads claiming a dead mutex will receive ownership with a special error code, `-EOWNERDEAD`. This serves as a warning: *the data protected by this mutex might be inconsistent, it should be fixed*. The new locker can do different things at this point:

- it can ignore it (scary choice!)
- it might be unqualified for the job and pass the responsibility on to somebody else (by unlocking).
- it can try to fix the data and succeed–then it will *heal* the mutex, setting its consistency state back to normal and proceed.
- or it can fail and pass it on...
- or it can fail and deem the state completely broken; to notify about this situation, it can mark the mutex *not-recoverable*, so all waiters and future claimers will get a `-ENOTRECOVERABLE` error code so other recovery strategies can kick in.

The most important aspect to take into account is that the user of the mutex has means to detect this situation instantly without having to rely on timeouts or other overheads.

1.4 Deadlock detection

A situation of deadlock happens when a task A that owns a mutex M tries to lock it again.

This is the simplest case, of course. The general case is:

- task T_1 owns mutex M_1 and tries to lock mutex M_2
- task T_2 owns mutex M_2 and is waiting to lock mutex M_3
- task T_3 owns mutex M_3 and is waiting to lock mutex M_4
- ...
- task T_N owns mutex M_N and is waiting to lock mutex M_1

Construct like these are called *ownership-wait chains*. And it is obvious that in this particular case, this chain would deadlock, as M_1 would never be released if T_1 is allowed to block waiting for M_2.

The only way to detect this situation is, upon lock time, to walk the chain and verify if the task that is about to lock owns any lock on the chain.

By definition this is a linear operation that is going to take time to execute. The best way to avoid this expensive check is to make sure our design uses proper locking techniques (like for example, acquire and release multiple locks in LIFO order).

2 The first try: rtfutex

Once the requirements were laid out, we first tried modifying the futex code in `kernel/futex.c`, adding functionality while maintaining the original futex interface.

In a glimpse, the locking mode used with futexes works like this (see [4]): there is a word in user space that represents the mutex. The fast lock operation is performed entirely in user space; if the word is unlocked, then it becomes locked and work proceeds. If it is locked, the program sets a different value in the user space word and then goes down to the kernel and waits.

When the unlock operation is performed, the unlocker will check the value of the word; if it indicates that only a fast-lock was performed (and thus there are no waiters in the kernel), it will be simply unlocked in user space; otherwise, it will ask the kernel to wake up one or more waiters. These waiters will come back to user space and reclaim the lock; only one will get it, the rest will go back to the kernel to sleep[6].

With this in mind, we performed the following modifications:

- To allow wake-the-highest-priority waiter behavior on a bound time, the hash table model had to be modified.

  One node per waiter was replaced by one node per futex, and each node would have its own priority-ordered list of waiters. Although the lookup of the futex-node in the hash table is $O(N)$, at least the manipulation of the waiter-list (or wait list) can be made $O(1)$.

  This introduced the need of having to allocate the futex-node, as it could not live in the stack of some waiter[7].

- In order to support robustness, deadlock detection and priority inheritance, the

[6] note this means that the lock is actually unlocked for an unspecified amount of time in an unlock to lock transition.

[7] This raises extra issues; allocation can fail and is not time-predictable; it can be slow, so it is needed to cache the nodes (as normally they are frequently reused); caching means a strategy is needed to purge them (garbage collection).

concept of *ownership* had to be added to the futex. This would save *which* task owns the futex on each moment. It also required to note in the task struct which futex was being waited for, as well as a list of owned futexes.

- A different method had to be used for locking in user space, the fast lock and unlock paths.

  The user space word representing the futex would store the PID of the locker while on the fast path and indicate with a bit the presence of waiters in the kernel. This way if a locker died after having done a fast-lock operation in user space (and thus the kernel not having any notion of it), a potential waiter could check if the lock was stale[8]. When a futex went into the *dead-owner* or *not-recoverable* state, the kernel would modify the user space word with special values to mark these states.

  As well, the unlock operation had to always be serialized, with the kernel assigning ownership and modifying the user space word, ensuring robustness[9] and that no lock stealing happened.

This design (and its implementation) was broken: the futexes are designed to be queues, and they cannot be stretched to become mutexes– it is simply not the same. The result was a bloated implementation.

As well, the code itself missed many fine (and not so fine) details:

- it suffered from race conditions: the modification of the different back pointers in the task struct was being done without protection.
- the priority inheritance engine was very limited (to the most simple cases of inheritance) and it didn't support `SCHED_NORMAL` tasks.
- serialized unlocking is slow, it causes the convoy phenomenon, and the code did not provide flexibility to allow the user to balance performance vs. robustness or priority inversion protection depending on the situation.
- it didn't provide the functionality at the kernel level, for usage by kernel code.
- it didn't support changing the priority of a task while it was waiting for a futex while at the same time properly repositioning it on the wait list according to its new priority.

While broken, it was perfect as a prototype– it gave an indication of what was wrong, how things should not be done and hinted which methods were a good idea. It was time to rethink all over again.

3 Trying again: fusyn

With rtfutexes we found that stretched designs are not a good idea, however, experience tells layered designs are a better idea.

The fusyn design follows the same basic principles of the futexes, providing the same service in kernel and user space. Enforcing a strict modularity among the different units that comprise it, it is possible to accomplish much more with less bloat and complexity.

[8] This is a very simple method that cannot guarantee conflicts when PIDs are reused; we implemented a naive task-signature system to try to avoid this case. We didn't realize how broken it was until later.

[9] If waiters coming up from the kernel died before locking again and there were still some others waiting, the kernel would never know about it and the remaining tasks would wait for ever.

The four main blocks that comprise the fusyn architecture are:

- *fuqueues* are the wait queues (very similar to the Linux kernel's waitqueues) and are the basic building block.
- *fulocks* provide the mutex functionality by adding the concept of ownership on top of fuqueues and dealing with all the priority promotion.
- *vlocators* serve as the link between user space words and the kernel objects (fuqueues and fulocks) associated to them.
- *vfulock sync* maintains synchronization between the fulocks and the *vfulocks*, the user space word associated to them. It also is responsible for identifying owners from the cookies stored in the vfulocks.

fuqueues

We start with a simple queue structure, `struct fuqueue`, declared in `linux/fuqueue.h`. It merely contains a priority-sorted list where to register the waiters for the queue, a spinlock and an operations pointer. The operations are for managing a reference count (used when associated to user space), for canceling a task's wait on a fuqueue and notifying the fuqueue of a priority change on a waiter (most functions are defined in `kernel/fuqueue.c`).

A fuqueue can be initialized, waited on with `fuqueue_wait()` or a number of waiters for it can be woken up with `fuqueue_wake()`. All the functions for doing that are conveniently broken up so they can be used by other layers.

Whenever a task waits on a fuqueue, it registers itself by filling up a `struct fuqueue_waiter`; that structure and the fuqueue being waited for are linked to from the task struct (`struct fuqueue_waiter *fuqueue_waiter` and `struct fuqueue fuqueue_wait`), so that the signal delivery code (through `fuqueue_waiter_cancel()`) and the scheduler priority changing functions (through `fuqueue_waiter_chprio()`) can properly locate which fuqueue to act upon. A spinlock protects these pointers in the task structure.

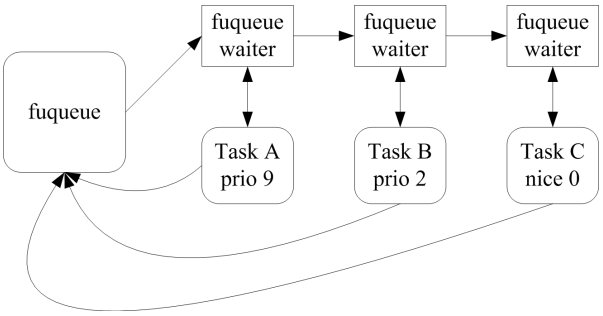

Figure 5: A fuqueue with three waiters, $p(A) > p(B) > p(C)$, showing the different pointers on each structure.

This satisfies the real-time requirements of wake-up order by priority. As well, the addition to the waiters list is bounded in time to the maximum number of different priority levels used–being this 140 for the Linux kernel, that makes the addition operation $O(140) \equiv O(1)$.

Note the fuqueue structure has to be protected, similarly to waitqueues with an IRQ-safe spinlock, as they will be accessed for wake-up from atomic contexts.

fulocks

Once we have a queue structure that is real-time friendly, we can build mutexes on top of them. Adding the concept of ownership, we create a `struct fulock` in `linux/fulock.h` that contains a fuqueue (for the waiters), a

pointer to a task struct (the owner), some flags and a node for a priority-sorted ownership list (to register all the fulocks owned by a task).

Let's ignore for a while the secondary effects of priority inheritance and protection. Come lock time, `fulock_lock()`: if the fulock is unlocked, the current task is assigned ownership by setting the owner pointer in the fulock to point to the task, the fulock is added to the `task->fulock_olist` ownership list through its `olist_node`.

If the fulock is locked (unless just try-locking) the task waits on the fulock's fuqueue; when woken up, depending on the result code of the wake up, it will own the lock (and thus proceed) or try again (serialized vs. parallelized unlocks).

The unlock operation, `fulock_unlock()` is quite simple: if the unlocker desires to perform a serialized wakeup, it just changes the owner to be the first waiter, removes it from the wait list and wakes him up with a 0 result code. If the unlock has to be parallelized, it unlocks the fulock and unqueues and wakes up the first waiter (or the first N waiters) with a `-EAGAIN` code–that will lead the sleeping `__fulock_lock()` call to retry. The unlock mode can be automatically determined based on the policy of the first waiting task: serialized for real-timers, parallelized for timesharers.

All this code is defined in `kernel/fulock.c`.

Robustness

Robustness comes into play with a hook in `kernel/exit.c:do_exit()`. When a process dies, `exit_fulocks()` goes over the list of fulocks owned by the exiting task; for each of them, the operation registered for task `exit` is executed, and that leads to setting the dead flag (`FULOCK_FL_DEAD`) and serially unlocking the fulock to the next waiter with the `-EOWNERDEAD` error code[10].

This introduces the need to have a way for the user to switch the fulock from one state to the other. `fulock_ctl()` provides this capability.

Deadlock detection

The process of checking for deadlocks is done via a hook in the `__fulock_lock()` function that calls `__fulock_check_deadlock()`.

This function will query the owner of the fulock the current task wants to wait for and inquire which fulock this owner is waiting for. If not waiting for anyone, there is no possible deadlock, so all resources are dropped and success is returned.

If it is waiting, the fulock is safely acquired (the ugliest part is to get the spinlocks properly as well as the reference counts); if the owner is the current task, then that is a deadlock; if not, then the operation repeats with the owner of the new fulock.

Priority inheritance and protection

Now let's take priority inheritance and protection into consideration. The key here is that in the priority-sorted list (plist), every node, including the list head, has a priority field, and that in a consistent plist, the priority of the list is that of the head, that in turn is that of the highest priority node queued.

Thus, by virtue of the priority-sorted list, each fuqueue has a *priority*. Fulocks inherit this property and when doing **priority inheritance**, they set that priority on the node for the priority-based ownership list. **Priority protected** fulocks set as priority that of the priority

[10] as well, a warning is issued if the fulock wasn't declared robust.

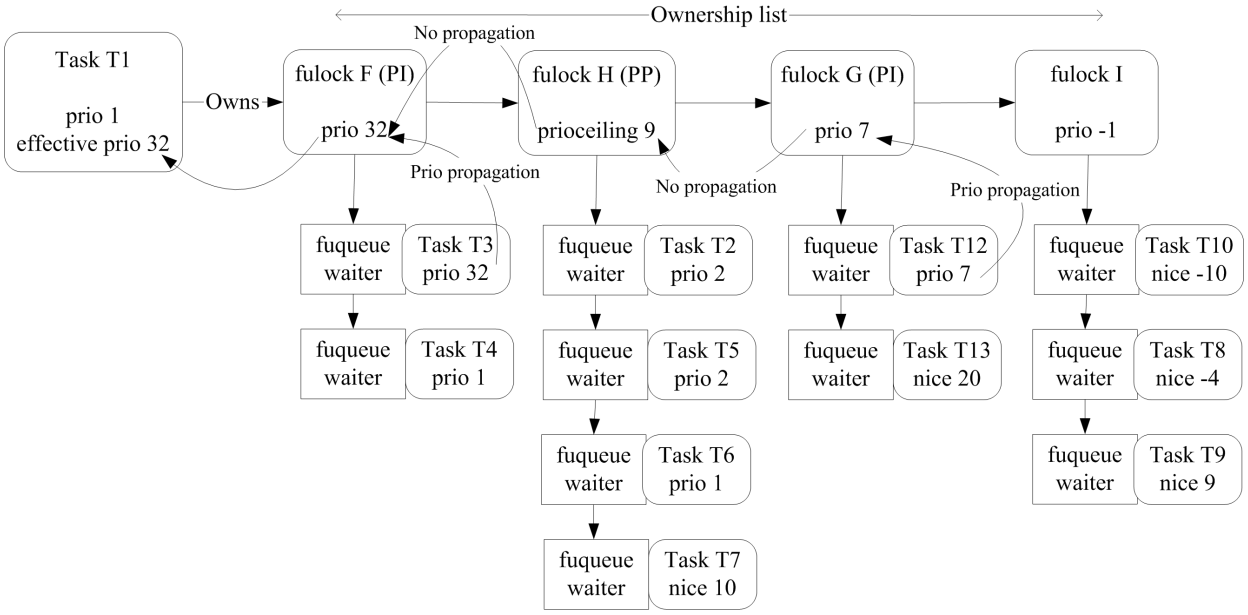

Figure 6: A task that owns four contested fulocks (two PI, one PP and one normal) showing the priority propagation flow. Note how fulock I, not being priority inheriting or protecting, has the minimal priority, -1 (which effectively disables all side effects).

ceiling of the fulock.

This way, each task has a list of the fulocks it owns sorted by priority. The ordering of the list means that the first fulock in the list has the minimum priority the task should have to meet the priority protection and/or priority inheritance criteria–and thus, the scheduler just has to select as effective task priority the highest between the task's final dynamic priority and that of the first fulock on its ownership list[11]. See Figure 6.

The process then becomes extremely simple: when a task queues waiting for a fulock (in `__fuqueue_waiter_queue()`), it might modify the plist priority because it sets a new *higher* priority–the function returns !0 in this case. This is propagated, with `__fulock_prio_update()` to the fulock's ownership list node, `fulock->olist_node`, that as we said above, is inserted in the ownership list of the fulock owner. The propagation could mean that a new maximum might be set in the ownership list, case in which the boost priority is updated for the scheduler to pick it up.

On top of that, the change might need to be propagated further on if the fulock owner is waiting for another fuqueue or fulock. `__fuqueue_waiter_chprio()` will take care of propagating that change until a task is reached that is higher priority or is not waiting for a priority-inheriting fulock.

Linking to user space

So far, the infrastructure presented is accessible only from kernel space. We have to allow user space programs to take advantage of these features, and for that, we copy the futex's method: associate a virtual address (word) to

[11] This is accomplished with a simple mechanism (improvement required to reduce invasiveness) that adds the concept of boost priority to the task struct (`boost_prio`), and modified through `__prio_boost()`.

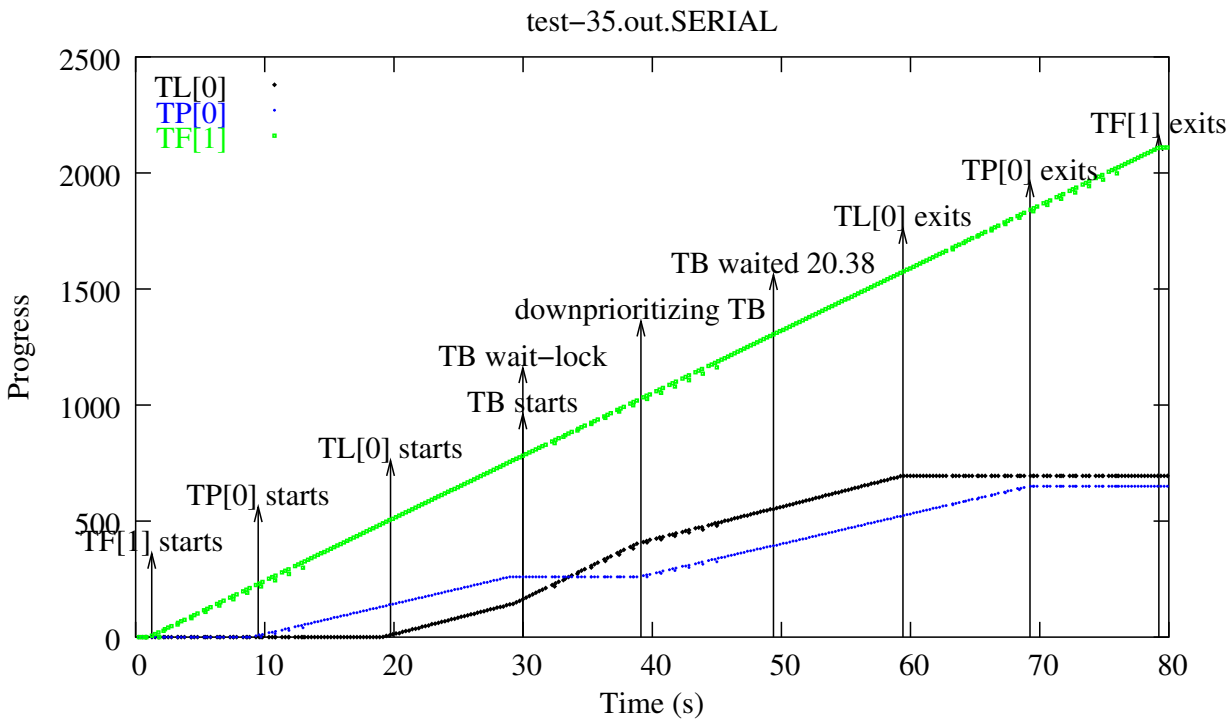

Figure 7: Testing priority inheritance: four threads of increasing priority (TL < TP < TB < TF) in an infinite loop counting up (progress); TF stays in CPU1 as a reference; TP sleeps from time to time in CPU0 to give TL a change; TL progresses what TP allows it. When TB starts (at $30s$), it claims a priority-inheriting fulock owned by TL and thus it gets boosted, TP doesn't progress any more. At almost $40s$, TB is down prioritized and that deboosts TL, allowing TP to progress again.

an object in memory (`struct vlocator`).

The API exposed in `linux/vlocator.h` provides a generic method for doing this by just embedding a vlocator; as well, this vlocator provides a reference-counting interface to simplify the object's life cycle management. And when it's use count is zero, it will be automatically disposed of[12].

This also improves scalability a little bit as the only global lock in the vlocator hash table is taken just to do the look up; once found, the vlocator is referenced before dropping the lock.

[12]Here is where the caching kicks in; the hash table is cleaned up of zero ref-counted items every certain amount of time, allowing for reuse.

ufuqueues and vfuqueues: imitating futexes

We need to create an interface equal to that of futexes for implementing conditional variables with real-time friendly functionality (for the wake up ordering).

We create a `struct ufuqueue` where we embed a vlocator and a fuqueue. A thin adaptation layer (`sys_ufuqueue_wait()` and `sys_ufuqueue_wake()`) will get the system call from user space, do the look up using the vlocator API, verify that the user space word (`vfuqueue`) hasn't changed and pass it down to the fuqueue layer.

The rest of the code in `kernel/ufuqueue.c` deals with creating the operation functions for

the vlocator structure.

Exposing the fulocks to user space

The same mechanism is used for exposing the fulocks to user space; in a similar fashion to fuqueues, we wrap together a vlocator and a fulock to create a `struct ufulock`.

However, more aspects have to be taken into consideration:

- If the fulock is not contested, the lock and unlock operations must happen entirely in user space (and thus the kernel will not know about it; this is the *fast-path*)
- When a lock has been locked through the fast path, the kernel has to be able to identify who locked it as well as its consistency status; this operation is called synchronization.
- When a lock becomes contested, the kernel has to update the user space word to indicate that future operations need to proceed in the kernel–as well, when it is eligible to be a fast-path only fulock again, the kernel must undo this, put the fulock structure in the cache tagged as requiring synchronization from user space and make sure the user space word has the consistency state of the fulock.
- The fulock structure in the kernel will be disposed if no task goes to the kernel querying about or operating on it for a while; as in the previous case, the information will be kept in user space word to enable proper synchronization.

For this we need some more information than the one used by the same futex mechanism for the fast path. A locker needs to identify itself in the user space word (that we call *vfulock*) by storing a cookie that can directly map to a task struct in the kernel space. The most obvious choice for the cookie would be the PID[13].

However, this operation must be atomic–this means that we need an atomic compare-and-exchange operation, and thus, the lock operation becomes the following: compare-and-exchange the cookie against 0 (meaning unlocked); if it succeeds, then the vfulock is locked, if not, dive into the kernel. The kernel will map the address to a fulock (possibly creating a new ufulock) get the value of the vfulock (`sys_ufulock_lock()` and `ufulock_lock()`) and map it to a task (in `__vfulock_sync()`. If the kernel is able to find the task, that task is made the owner and the caller is put to wait. As well, the vfulock is updated to a special value `VFULOCK_WP`, meaning waiters are present in the kernel.

If the kernel cannot find it, that will mean the task that fast-locked it in user space has died, the fulock will be declared *dead-owner* and the caller will get ownership. In this process, the vfulock will be set to another special value, `VFULOCK_DEAD` that indicates it as dead even across the kernel forgetting about its existence.

Unlocks are equally simple: atomically compare-and-exchange 0 (`VFULOCK_UNLOCKED`) against the cookie of the lock owner; if it succeeds, the job is done; else, the kernel does it. After mapping the vfulock to a ufulock, `ufulock_unlock()` is used to do the job and the vfulock is updated to reflect the new state: `VFULOCK_UNLOCKED` if unlocked, if there will be no waiters the new owner's cookie–enabling fast-path, `VFULOCK_WP` if waiters are still in the kernel, or `VFULOCK_DEAD` if the fulock is dead.

If parallelized unlocks are desired, the pro-

[13] This would break unique identification as PIDs are reused; a solution could be crypting the PID with the task creation date, but it needs to be tested.

cess is a little bit different. In the kernel, `__ufulock_unlock()` will unlock the vfulock and then wake up the first waiter, who then will contend (in the kernel) for the vfulock and possibly wait, as described above[14].

Note the two key moments in switching from fast-path enabled or not: the fulock becomes fast-path when it has no waiters in the kernel or when it is healed[15] without waiters. It looses the fast-path conditions as soon as a single waiter is queued. This means that to maintain proper semantics during the lifetime of a program that uses many locks, once a fulock has gone through the slow path, it needs to be destroyed in the kernel using the `sys_ufulock_ctl()` system call once it is not needed anymore. If not, there could be inconsistencies if a new lock is created in the same address where a previous one lived before.

KCO: When the fast-[un]lock path is not an option

The fast path, as we have seen, requires an atomic compare-and-exchange operation. Not all architectures provide this capability, so different strategies need to be considered here.

If robustness, and priority inversion protection[16] can be spared, the mutexes and conditional variables can be implemented as with futexes using fuqueues; the rest of the real-time featurettes are there (priority-based wake-ups and priority change semantics). If that can also be spared, futexes are still an option.

However, when that is not the case, the only possible choice is to use KCO mutexes, by ORing `FULOCK_FL_KCO` in the flags. That is an acronym for Kernel Controlled Ownership, or basically, the kernel takes care of everything. It needs to be called for locking and unlocking, there is no fast path (strictly speaking there is still a choice for fast path on some operations, as the vfulock is used to cache the consistency state of the fulock and any user space operation can check it before deciding if it should go to the kernel).

This feature also provides the highest level of protection for robustness. The per-thread cookie for the vfulock, be it the PID or any other, is not required, and the kernel deals directly with the task struct, so there is no possible collision conflict.

It has to be noted that priority-protected ufulocks always work in KCO mode. Even on uncontended acquisition or release the priority of the thread has to be changed to that of the prioceiling, and that task can only be done by the kernel.

4 Using it in the kernel

The fulock is a simple type like any other struct. To use it, we just need to do the following declarations:

```
...
#include <linux/fulock.h>
...

struct mystruct {
  struct fulock lock;
  ...
  my shared data;
};
```

It needs to be properly initialized before use, and of course, after releasing it (or more properly, telling all waiters to bail out) it shall not

[14] Not going back to user space to retry the operation has advantages: speed and maintaining the conditions for robustness.

[15] Moved from *dead-owner* consistency state back to normal (or healthy)

[16] Lock stealing avoidance, priority inheritance and priority protection.

be used. Note some flag combinations are not allowed (for example, querying for priority inheritance and protection at the same time is illegal) and will trigger a `BUG()`[17].

In this example we ask for a robust fulock with priority inheritance. It must be noted that fulocks are always robust–but clearly telling the kernel that we handle robust situations will suppress a kernel warning if the owner dies and it goes into *dead-owner* mode.

```
my_driver_probe(...)
{
 struct mystruct *my;
 ...
 my = kmalloc (...);
 if (my == NULL)
  goto err_alloc;
 fulock_init (&my->lock,
              FULOCK_FL_ROBUST
              | FULOCK_FL_PI);
 ...
};
```

As we see in the following snippet, the basic usage is the same as for every lock. However, in this case we add some recovery code for the case when some owner died[18]. Note also that the only fulock operation that is guaranteed to be safe in an atomic context is `fulock_unlock()`.

```
void my_something(
 struct mystruct *my) {
 ...
 result = fulock_lock(&my->lock,
                      0);
 if (result == -EOWNERDEAD
     && my_try_recover (my))
   goto notrecoverable;
 ...
 /* do our thing */
 ...
 fulock_unlock (&my->lock,
                FULOCK_FL_AUTO);
 ...
 return 0;

notrecoverable:
 /* Put it out of its misery,
  * release waiters, clean up,
  * user has to reload the
  * driver. */
 fulock_ctl (&my->lock,
             FULOCK_CTL_NR);
 my_put (my);
 return -ENOTRECOVERABLE;
};

int my_try_recover (struct
    *mystruct my) {
 int result, mode;
 ... try to recover *my ...
 if (successful) {
  result = 0;
  mode = FULOCK_CTL_HEAL;
 }
 else {
  result = !0;
  mode = FULOCK_CTL_NR);
 }
 fulock_ctl (&my->lock, mode);
 return result;
}
```

Finally, when we are done, we release all resources associated to the fulock to clean up. As indicated above, this merely makes sure that any waiter queued is woken up with an error condition and nobody can acquire it or queue again.

```
void my_cleanup (
 struct mystruct *my)
{
 ...
```

[17] For user space code, they will simply fail with `-EINVAL`.

[18] This is kind of an useless exercise, correct kernel code doesn't crash.

```
 fulock_release (&my->fulock);
 ...
}
```

The benefits that a fulock gives over a semaphore are the real-time characteristics, priority inheritance and protection and deadlock detection. The decision to use one or the other depends on the user needs, as it has to be taken into account that fulocks are somehow more heavyweight than semaphores.

5 Usage from user space

The main intention of the user space code is to do as little as possible in the fast path and delegate the rest to the slow path that will, in most cases, end up in the kernel.

Note these code snippets have been slightly simplified; for the authoritative reference, see the file `src/include/kernel-lock.h` in the test package `fusyn-package` available from the web site.

Locking

As mentioned, the fast lock operation needs an atomic compare and swap operation; for example, on i386:

```
unsigned acas (
 volatile unsigned *value,
 unsigned old_value,
 unsigned new_value)
{
 unsigned result;
 asm __volatile__ (
  "lock cmpxchg %3, %1"
  : "=a"(result),"+m"((*value))
  : "a"(old_value),"r"(new_value)
  : "memory");
 return result == old_value;
}
```

To simplify the code, this function returns true if it was successful in performing the swap operation. With this, we can create a generic, fast-path, user space lock operation:

```
int vfulock_timedlock (
 volatile unsigned *vfulock,
 unsigned flags, int pid,
 struct timespec *rel)
{
 if (acas(vfulock,
          VFULOCK_UNLOCKED,pid))
  return 0;
 return SYSCALL (ufulock_lock,
                 vfulock, flags,
                 rel);
}
```

We are using the thread's PID as the cookie for the vfulock, the user space memory word associated to the lock. Note the special syntax for timeouts understood by the kernel:

- Passing `NULL` means we don't want to wait, and this operation effectively becomes a trylock in the kernel.
- A `(void *)-1` timeout means block forever–no timeout.
- Any other specifies a pointer to a valid timeout structure.

From user space we have to always pass the same flags to the kernel for an specific vfulock, as it will check we are consistent during the lifetime of the fulock–when it dissapears from the cache, it is up to us to use still the same flags to maintain consistency in our program.

With a few additions, we can have a lock function that also works in KCO mode and that imitates the behavior of non-robust mutexes when owners die (*ie:* block forever):

```
int vfulock_timedlock (
 *vfulock, flags, pid, *rel)
{
 int result;
 if (!(flags & FULOCK_FL_KCO) &&
     acas(vfulock,
          VFULOCK_UNLOCKED,pid))
  return 0;
 result = SYSCALL (ufulock_lock,
                   vfulock, flags,
                   rel);
 if (!(flags & FULOCK_FL_RM) &&
     (result == -EOWNERDEAD
     |result == -ENOTRECOVERABLE))
  waiting_on_dead_fulock(vfulock);
 return result;
}
```

There are only two simple differences. First is to avoid the fast-path if we want to use KCO mode (and thus dive directly into the kernel). The second one takes care of non-robust mutexes returning in *dead-owner* state; in that case we block in `waiting_on_dead_fulock()`, a dummy function that blocks forever whose only purpose is to show up in program traces to indicate us the reason of a thread blocking.

Unlocking

The unlock operation is somehow more hairy. Although we could just make it simpler calling the kernel and letting it do all of the operations for us (as if it were in KCO mode), we want to have the fast-unlock path available:

```
int __vfulock_unlock (
  *vfulock, flags, unlock_type)
{
 unsigned old_value = *vfulock;

 if (flags & FULOCK_FL_KCO)
   goto straight;
retry:
 if (old_value < VFULOCK_WP) {
  if (acas (vfulock, old_value,
            VFULOCK_UNLOCKED))
   return 0;
  old_value = *vfulock;
  goto retry;
 }
straight:
 return old_value == VFULOCK_NR?
  -ENOTRECOVERABLE
  : SYSCALL (ufulock_unlock,
             vfulock, flags,
             unlock_type);
}
```

As with the lock() operation, we first check if the fulock is KCO; if so jump straight into the kernel (except if it is marked *not-recoverable*, in which case we fail).

In the case of the fast-path, we read the value of the vfulock; if it is looks like a cookie[19] then we try the fast-unlock, returning if successful. If it failed we retry from the beginning. When the value of the vfulock doesn't look like a cookie, we dive into the kernel, as it means that it is either dead or there are waiters (and thus the kernel handles it).

Note this unlock operation allows any thread to unlock the fulock, it doesn't need to be the owner.

Other operations

A `trylock()` operation is implemented in similar terms (please refer to the sample library code in the `fusyn-test` package, file `src/include/kernel-lock.h`; this package is available for download from the project's website).

Operations for manipulating or querying the

[19] The three values `VFULOCK_WP`, `VFULOCK_DEAD` and `VFULOCK_NR` are purposely chosen to be the last three values of the `unsigned` domain.

state of the fulock are implemented by calling the `ufulock_ctl()` system call directly, providing the vfulock and flags.

6 Integration with NPTL

The patches for integration with NPTL (that we call RTNPTL for short) allow any POSIX program to use these features, via a certain set of standard calls and ways to customize the operation mode of the fulock under the mutex's hood with other non-POSIX extensions.

RTNPTL uses the same or very similar user mode integration code than the one explained above, sitting down at the `lll_` layer in glibc. This code provides all the intended functionality only to the POSIX mutexes and conditional variables. Locks used internally by the library still need work (see the *future directions* section).

By default, RTNPTL provides non-robust fast-path enabled mutexes that unlock in automatic mode[20], without any priority inheritance and protection. However, by modifying the mutex attributes with the `pthread_mutexattr_set*()` calls, different parameters can be set:

- Manipulating the priority inversion protections:

  `pthread_mutexattr_setprotocol()` takes a mutex attribute and a protection protocol, `PTHREAD_PRIO_INHERIT` or `PTHREAD_PRIO_PROTECT`.

  `pthread_mutex_setprioceiling()` can be used to query and change the priority ceiling of a mutex.

  `pthread_mutexattr_setserial_np()` and `pthread_mutex_setserial_np()` allows setting the unlock method to use for lock-stealing avoidance out of `PTHREAD_MUTEX_SERIAL_NP`, `PTHREAD_MUTEX_PARALLEL_NP`, or `PTHREAD_MUTEX_AUTO_NP` (this one can be switched during the lifetime of the mutex).

- `pthread_mutexattr_setrobust_np()` enables robustness in the mutex to be. `pthread_mutex_setconsistency_np()` is used to heal or make *not-recoverable* a *dead-owner* mutex. The consistency state can be queried with `pthread_mutex_getconsistency_np()`.

- `pthread_mutexattr_setfast_np()` is used to select the use of a KCO fulock or not, effectively enabling/disabling fast-path operation.

The non-standard interfaces are still subject to some unlikely flux.

7 Current status and future direction

At the time of writing, the project has met most of the requirements that were set as targets, reaching stability and meeting performance goals of sub-millisecond latencies. The added overhead does not seem to affect too much compared to NPTL, being generally slightly slower.

Compatibility

We routinely test RTNPTL+fusyn by running:

[20] serialized or parallelized depending on the policy priority of the first waiter

- Miscellaneous multi-threaded applications (e.g.: Mozilla)
- SUN jdk-1.42_03 with SPECjbb2000[21].
- MySQL 2.23.58 with `super-smack` and `sql-bench`.

This has helped us to catch some bugs (with some pending for certain combinations) and to test the compatibility of our approach. Performance wise, no obvious differences have been found with plain NPTL running on futexes.

This set of macro benchmarks is incomplete and will be expanded in the future, time and resource availability permitting.

Latency

The current code performs fairly well latency wise (given the extra overhead). In an unloaded system[22], the latency of the serialized ownership change operation[23] is in the range of $60 \pm 10\mu s$. Adding some network load (ten simultaneous downloads of 40 MiB files) bumps it up to $110 \pm 10\mu s$. Simultaneous reading of 1 GiB from `/dev/hda` to `/dev/null` raises it up to $130 \pm 10\mu s$.

The code exposes a strange behavior when testing the ownership change latency in an unloaded system while increasing the number of waiters. The average latency stays stable for the first ten-to-fifteen waiters (threads of a single program) at around $18 \pm 10\mu s$ (see Figure 8).

However, when the number of queued waiters goes up to 2000 threads, the latency climbs up to $50 \pm 10\mu s$, stabilizing from there on, as seen on Figure 9. Of course this is an extremely unrealistic scenario, but it helps to test the scalability of the code, and nevertheless, we are trying to proof the root cause, being cache issues the most likely ones.

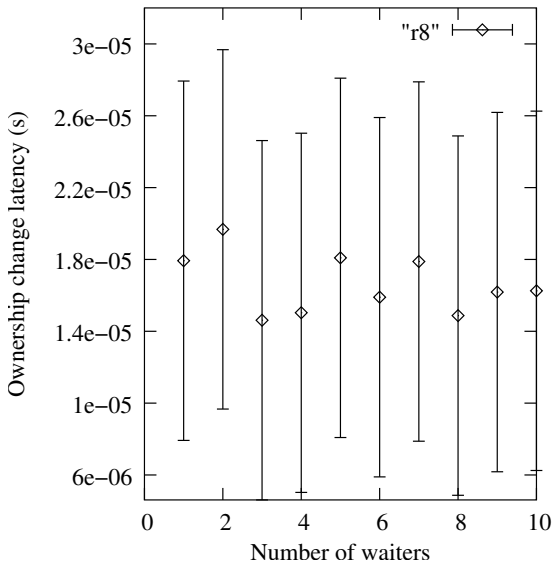

Figure 8: Scalability of the ownership-change latency *vs.* the number of waiters stays stable up until ten waiting threads.

.

Note: these numbers have been produced with a home-grown swiss-knife test program (to be published on the web site) called `ownership_change_latency`. Most of our timing efforts have concentrated in this particular case, although we have some other micro benchmarks planned.

Jitter

At this point, we haven't done yet any formal jitter studies.

Informally speaking, using the ownership change latency benchmark in unloaded systems, we have seen jitter increases over NPTL of about $1\mu s$, $0.3\mu s$ on a system fairly loaded with IDE and network traffic. However, bear in

[21] SPEC Java Business Benchmark 2000.

[22] as measured in a 2xP3 850 MHz 2.5 GiB RAM running version 2.3 of the code

[23] time since a serialized unlock is done until the first waiter gets the lock and executes.

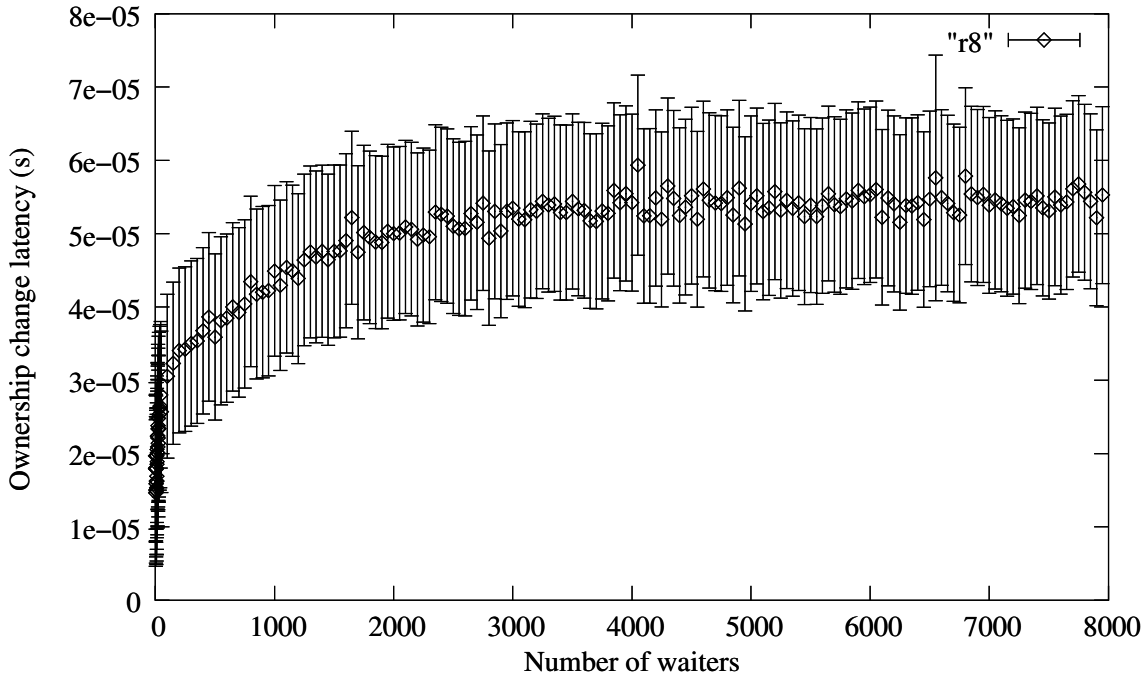

Figure 9: Scalability of the ownership-change latency *vs.* the number of waiters only stabilizes after two thousand waiters.

.

mind that these numbers are completely meaningless because the finest dependable clock resolution we can get (using the High Resolution Timers patch) is well higher, $10\mu s$. We can use them only to provide a hint.

Future direction

The project has reached an important milestone of maturity with the 2.2 release during the spring of 2004–nonetheless there is still much work to do. These some of the areas where we plan to target our future efforts:

- Some parties have asked for all these concepts (real-time, robustness, priority-protection) applied to read-write mutexes, much more complex than simple mutexes. We are still evaluation how worth is this.
- Some elusive bugs are still present.
- Accessing user space memory from the kernel by `kmapping` it poses some issues on architectures with *strange* cache consistency designs, such as some ARM and PA-RISC 8000. It is still not clear how to proceed for them and we would welcome any help.
- The kernel hash table for location of objects is a potential bottleneck in a system populated with many active user-space fusyn objects. We want to implement a proof of concept where a cookie identifying the object is placed in user space along the vfulock/vfuqueue. This cookie would consist of a two pointers crypted with two different keys by the kernel. In order to map a vfulock/vfuqueue to it's corresponding fusyn object, the kernel just has to decrypt the pointers. Having two crypted with different keys is used to enforce validity against garbage being writ-

ten by user space by mistake or to compromise the system.

- Providing a robust mutex infrastructure is OK, as long as it is used. Internally, glibc uses locks to protect many of its data structures—in order to be able to provide true robustness, we need to add robustness to those internal locks, as well as recovery strategies.

- Extend the coverage of our macro and micro benchmarks.

8 Downloading

The project maintains a website at:

`http://developer.osdl.org/dev/robustmutexes/`

from where all the current and older snapshots of the code can be obtained. As well, it offers pointers to the mailing list, bugzilla and CVS repositories.

We want to thank the Open Source Development Lab for making these resources available to us.

9 Conclusion

We have presented an infrastructure for providing real-time and robust synchronization services in the Linux kernel. We have been able to accomplish this with a minimum overhead impact over the current futex-based infrastructure and expect that it will be sufficient to satisfy the needs of multi-threaded, fault-proof and/or soft-real time designs.

10 Trademarks and acknowledgements

During development we kept discovering roadblocks, situations, and side effects we failed to spot or details we missed in the POSIX specifications. A lot of hair pulling that was counteracted by the thrill of the challenge, producing a love-and-hate relationship with the topic (and hence the title of this paper). We want to thank all of those who helped out by pointing out issues, contributing, reviewing, criticizing, and testing ideas and code.

References

[1] Mike Blasgen, Jim Gray, Mike Miltoma, and Tom Price. 1979. “The convoy phenomenon,” *ACM SIGOPS Operating Systems Review*, Volume 13, Issue 2 (April 1979). `http://portal.acm.org/citation.cfm?id=850659&jmp=cit&dl=GUIDE&dk=ACM`

[2] Arnd C. Heursch, Dirk Grambow, Dirk Roedel, and Helmut Rzehak. “Time-critical tasks in Linux 2.6,” `http://www.informatik.unibw-muenchen.de/inst3/index_de.php` Pages 6 and 7.

[3] Victor Yodaiken. 2001. “The dangers of priority inheritance,”

http://citeseer.ist.psdu.edu/yodaiken01dangers.html

[4] Hubertus Frankel, Rusty Russell, and Matthew Kirkwood. 2002. “Fuss, Futexes and Furwocks: Fast Userlevel Locking in Linux.” *Proceedings of the 2002 Ottawa Linux Symposium*, http://www.linux.org.uk/~ajh/ols2002_proceedings.pdf.gz Pages 479-495

Proceedings of the Linux Symposium

Volume Two

July 21st–24th, 2004
Ottawa, Ontario
Canada

Conference Organizers

Andrew J. Hutton, *Steamballoon, Inc.*
Stephanie Donovan, *Linux Symposium*
C. Craig Ross, *Linux Symposium*

Review Committee

Jes Sorensen, *Wild Open Source, Inc.*
Matt Domsch, *Dell*
Gerrit Huizenga, *IBM*
Matthew Wilcox, *Hewlett-Packard*
Dirk Hohndel, *Intel*
Val Henson, *Sun Microsystems*
Jamal Hadi Salimi, *Znyx*
Andrew Hutton, *Steamballoon, Inc.*

Proceedings Formatting Team

John W. Lockhart, *Red Hat, Inc.*