Requeue-PI: Making Glibc Condvars PI-Aware

Darren Hart
International Business Machines
15300 SW Koll Pkwy, DES2 4F 1, Beaverton, OR 97006, United States
dvhltc@us.ibm.com

Dinakar Guniguntala
International Business Machines
Embassy Golf Links, Koramangala Ring Road, Bangalore, KA 560071, India
dino@in.ibm.com

Abstract

Glibc 2.5 and later provide some support for priority inheritance (PI), but some gaps in the `pthread_cond*` APIs severely cripple its utility and lead to unexpected priority inversions and unpredictable thread wake-up patterns. By adding kernel support for proxy locking of rt-mutexes and requeueing of tasks to PI-futexes, glibc can provide PI support across the entire spectrum of mutex and condvar APIs. For some complex applications, completely avoiding priority inversion can be impractical if not impossible. For such applications, this provides a robust POSIX threading mechanism. Broadcast wake-ups can now be more efficient, waking threads up in priority order as the contended resource becomes available.

1 Introduction

Condition variables (condvars) provide a convenient userspace API to block a thread (or group of threads) on an event. Under the covers, condvars are implemented using futexes. A condvar uses multiple futexes, one for use as a wait-queue and one for locking its internal state. In addition, each condvar has an associated mutex specified by the waiters, which is also implemented using a futex. Threads blocking on an event call `pthread_cond_wait()` [1] with the associated mutex held (glibc will release the mutex prior to calling into the kernel), see Figure 1. When the waking thread calls `pthread_cond_broadcast()` [2], the caller expects the blocked threads to stop waiting on the event and begin contending for the associated mutex. The mutex is acquired before control returns to the application code (glibc performs the mutex acquisition). `pthread_cond_broadcast()` notifies all threads of the event (while `pthread_cond_signal()` only notifies one) [3] [11].

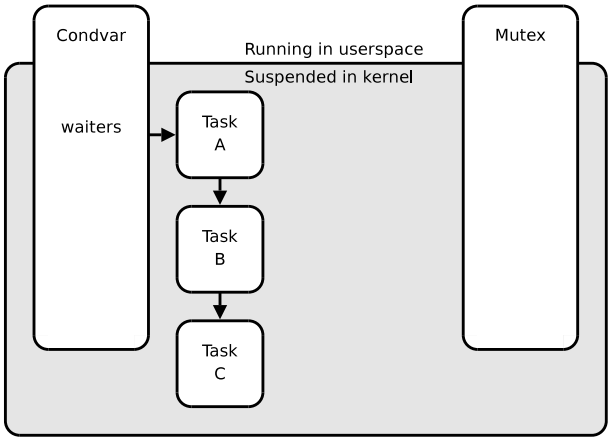

FIGURE 1: *Threads blocked on condvar after multiple pthread_cond_wait() calls.*

[1] Throughout the paper, when pthread_cond_wait() is mentioned, the same applies to pthread_cond_timedwait() unless stated otherwise.

[2] Throughout the paper, when pthread_cond_broadcast() is mentioned, the same applies to pthread_cond_signal() unless stated otherwise.

[3] See the man pages for more gory details on the POSIX standard.

In the event of a broadcast, only one thread is woken to return to userspace, while the rest are requeued from the condvar to the mutex [4]. The thread that is woken then acquires the mutex (in glibc) before returning to the application, see Figure 2. The unlocking of the mutex by the application then triggers the next eligible thread to be woken, and so on.

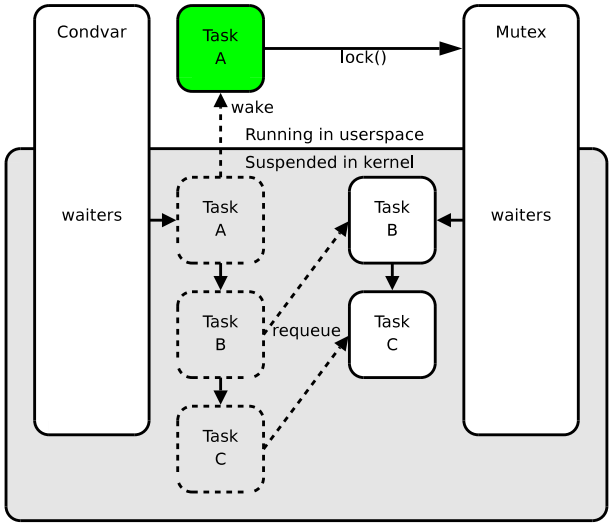

FIGURE 2: *Thread state after pthread_cond_broadcast() (non PI case).*

Futexes make use of the kernel-side `rt_mutex` for priority inheritance support. These locks have complex semantics regarding their owner and waiters which caused requeue support for PI futexes (requeue PI) to be deliberately left out in the initial implementations. Without kernel support for requeue PI, glibc could only wake all the waiters blocked on the condvar's wait-queue and leave them to race to return to userspace and try to acquire the associated mutex, see Figure 3. The first one to return got the mutex and the rest were put back to sleep on the mutex [5]. This effectively requeues all the threads but one, but does so in a very inefficient way. This causes additional context switches and compromises deterministic scheduling since no care is taken to ensure the highest priority waiting task acquires the mutex. This "wake all" approach creates what is commonly referred to as a "thundering herd".

Additionally, the internal lock protecting the condvar's state is not PI-aware. This means that even while using a PI-aware mutex, users of condvars are susceptible to unbounded priority inversion on the condvar's internal lock. An unbounded priority inversion can occur when a high priority task is blocked on a lock held by a lower priority task that has been preempted by a medium priority task. In this scenario, the high priority task is blocked for as long as there are runnable tasks of higher priority than the lock holder.

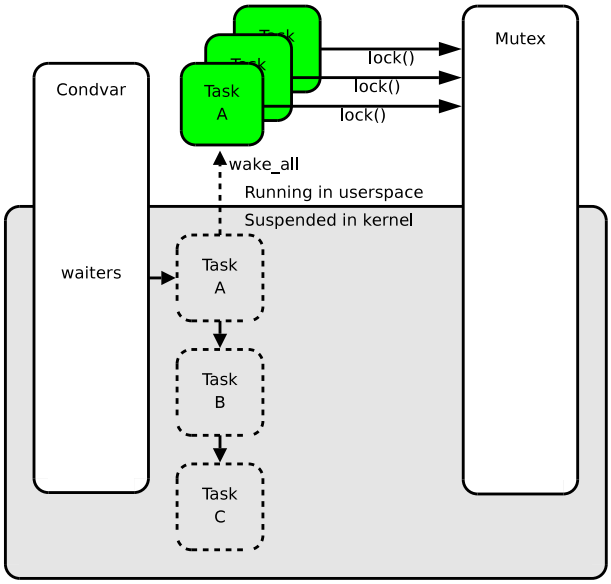

FIGURE 3: *Thread state after pthread_cond_broadcast() (PI case without requeue PI support).*

PI is often used in conjunction with other real-time features of the Linux kernel. These gaps in the PI stack hurt performance for applications using PI mutexes with condvars at best, and can lead to non-intuitive priority inversions at worst. The additional wake-ups, latency, and the unbounded priority inversion can be a show stopper for real-time applications with hard deadlines. Earlier attempts [14] to solve this problem were rejected for various reasons, including complexity and inefficiency. Nonetheless, those early attempts were valuable in guiding the development of this solution.

In the following sections, we will describe our solution to the problem, which consists of both kernel and glibc changes. Section 2 presents examples of test cases that fail or are adversely affected by this problem, as well as detailing the specific technical challenges involved in devising a solution. Section 3 provides an overview of the general Linux futex implementation as background for Section 4 which describes the Linux kernel and glibc aspects of the solution. Finally, Section 5 presents the successful results of the same test cases shown earlier.

[4] Assuming the mutex does not have the priority inheritance attribute set.

[5] This scenario is complicated slightly if the signaling thread holds the mutex across the call to pthread_cond_broadcast().

2 Problem Definition

2.1 Failure Scenarios

The disorderly wake-up is really a functional issue, especially when viewed in the context of real-time applications. The prio-wake test case from the LTP [6] real-time test suite illustrates the problem readily. Prio-wake creates `NR_CPUS` `SCHED_FIFO` threads of increasing priority and puts them to sleep on a condvar. It then calls `pthread_cond_broadcast()` and logs the threads' wake-up order. If they do not wake in strict priority order the test fails, as illustrated in Listing 2, lines 12-13. The test can be run with or without the associated mutex held across the `pthread_cond_broadcast()` call. It is next to impossible for a current system to pass the test without the lock held, but even with it held, failures are very common (75% in our testing).

It is difficult to provide a non-contrived test case for unbounded priority inversion on the condvar internal lock. However, the problem has been observed on Real Time Specification for Java (RTSJ) [8] compliance test suites and other large thread-intensive applications. This problem usually manifests by the program not making progress with high priority threads blocked on a `pthread_cond_*` call.

2.2 No Ownerless RT-Mutexes

The mechanism used to implement PI within the Linux kernel is the `rt_mutex`. The `rt_mutex` maintains a list of tasks blocked on it in its `wait_list` field. Each task maintains a list of the highest priority task blocked on every `rt_mutex` it owns in the `pi_waiters` list of the `task_struct`. These lists combine to form the priority inheritance chain (PI chain) that is used to boost the priority of the `rt_mutex` owner to that of the highest priority task blocked on the `rt_mutex`. The details of the PI chain are beyond the scope of this paper, but are well documented in [2] and [15]. It is sufficient to state that to ensure the PI boosting logic succeeds whenever a new task attempts to acquire a contended lock, an `rt_mutex` must never be in a state where it has waiters and no owner. Without an owner, the PI boosting logic would not find a task to boost, and the PI mechanism would break down.

The original glibc implementation of `pthread_cond_wait()` for non-PI mutexes waits for the waiter to return from the kernel after a `pthread_cond_broadcast()` before trying to acquire the associated mutex, as illustrated by lines 7-9 of Listing 1.

The problem with this implementation is the window of time between the wake-up after line 7 and the lock on line 9. Using the futex requeue mechanism, the waking thread would wake one waiter and requeue the remaining to the associated mutex. It then returns to userspace and releases the associated mutex [6]. Meanwhile, the newly woken thread attempts to acquire the mutex. Should the waking thread release the mutex before the waiter can block on it, the mutex will be left with no owner and multiple waiters, violating the usage semantics of the `rt_mutex`.

```
1  cond_wait(cond, mutex)
2  {
3      lock(cond->__data.__lock);
4      unlock(mutex);
5      do {
6          unlock(cond->__data.__lock);
7          futex(cond->__data.__futex,
8                FUTEX_WAIT);
9          lock(cond->__data.__lock);
10     } while(...)
11     unlock(cond->__data.__lock);
12     lock(mutex);
13 }
```

Listing 1: Original pthread_cond_wait() pseudocode.

As mentioned previously, the original solution to this problem was to wake all the tasks blocked on the condvar, rather than requeue them as waiters, and allow them to race for acquisition of the associated mutex. Once the mutex is acquired by one of the waiters, the remaining late arrivals will then be put to sleep on the mutex. Future unlocks will work the same as any `pthread_mutex_unlock()` call would, giving the lock to the next highest priority waiter and waking it up. The initial race leads to unpredictable wake-up order and a lot of unnecessary context switches and thrashing about within the scheduler.

3 Meet the Futex

Prior to diving into the problems priority inheritance presents futexes, a basic understanding of what futexes are and how they work is needed. The best reference for the basic operations of futexes and how they can be used to create higher-level locking mechanisms is probably still Ulrich Drepper's "Futexes

[6] It is possible that the waking thread did not hold the mutex, which is a valid usage of the API, in which case it will not need to unlock it after signaling the waiters.

```
# ./prio-wake

-------------------------
Priority Ordered Wakeup
-------------------------
Worker Threads: 8
Calling pthread_cond_broadcast() with mutex: LOCKED

00000527 us: Master thread about to wake the workers

Criteria: Threads should be woken up in priority order
FAIL: Thread 7 woken before 8
Result: FAIL
000000: 00000102 us: RealtimeThread-8388768 pri 001 started
000001: 00000166 us: RealtimeThread-8389264 pri 002 started
000002: 00000217 us: RealtimeThread-8389760 pri 003 started
000003: 00000284 us: RealtimeThread-8390256 pri 004 started
000004: 00000332 us: RealtimeThread-8390752 pri 005 started
000005: 00000386 us: RealtimeThread-8391248 pri 006 started
000006: 00000432 us: RealtimeThread-8391744 pri 007 started
000007: 00000481 us: RealtimeThread-8392240 pri 008 started
000008: 00000691 us: RealtimeThread-8391744 pri 007 awake
000009: 00000753 us: RealtimeThread-8392240 pri 008 awake
000010: 00000813 us: RealtimeThread-8391248 pri 006 awake
000011: 00000855 us: RealtimeThread-8390752 pri 005 awake
000012: 00000885 us: RealtimeThread-8390256 pri 004 awake
000013: 00000929 us: RealtimeThread-8389760 pri 003 awake
000014: 00000953 us: RealtimeThread-8389264 pri 002 awake
000015: 00000968 us: RealtimeThread-8388768 pri 001 awake
```

Listing 2: Failing prio-wake example.

are Tricky" [12]. "Fuss, Futexes and Furwocks: Fast Userlevel Locking in Linux" [13] is also a good historical reference, although fairly out of date at this point.

A futex is a fast userspace lock [3], although there is truly no such structure that can be said to encapsulate a futex. A futex is a very basic locking mechanism, and is typically used to construct higher level primitives. The most basic component of a futex is its userspace variable, a four byte unsigned integer on all platforms. The uncontended path usually involves the futex variable being marked as locked in userpace without having to call into the kernel. While the futex implementation initially made no requirements on how this value is used, the priority inheritance implementation uses this value to determine the owner of the futex and whether or not it has waiters.

From the kernel side, each futex address is converted into a `futex_key` using the physical memory address so that processes sharing a futex can use their own address space to reference the same futex. Each task blocked on a futex is represented by a `futex_q` structure, see Listing 3, which uses the `futex_key` to add itself to a hash-table of priority lists of `futex_q` structures. As there are typically only 128 entries in the hash-table, collisions are expected; each hash-list will contain `futex_q` structures representing tasks waiting on multiple futexes. The `futex_key` is used to determine which elements are of interest to a given operation. Figure 4 illustrates this structure. A simple single-linked list is depicted for clarity, however, a `plist` is used in the actual implementation to ensure tasks are queued in fifo order by priority [15].

```
struct futex_q {
  struct plist_node list;
  struct task_struct *task;
  spinlock_t *lock_ptr;
  union futex_key key;
  struct futex_pi_state *pi_state;
  struct rt_mutex_waiter *rt_waiter;
  union futex_key *requeue_pi_key;
  u32 bitset;
};
```

Listing 3: struct futex_q.

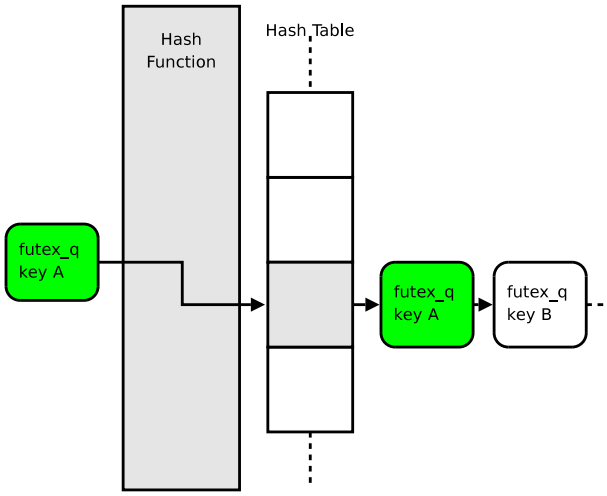

FIGURE 4: *Futex hash table queueing structure.*

Futexes are fast in the uncontended case, because the acquisition can be done from userspace using atomic instructions, such as `cmpxchg` on x86. In the contended case, the kernel is invoked to handle blocking and wake-up. There is a single system call for all futex related operations, `sys_futex`, shown in Listing 4. It accepts an operation argument, `op`, which is used to determine the appropriate kernel function.

```
long sys_futex(void *addr1, int op,
  int val1, struct timespec *timeout,
  void *addr2, int val3);
```

Listing 4: sys_futex definition

The supported futex operations include: `FUTEX_WAIT`, `FUTEX_WAKE`, `FUTEX_CMP_REQUEUE`, `FUTEX_LOCK_PI`, `FUTEX_UNLOCK_PI` (and now `FUTEX_WAIT_REQUEUE_PI` and `FUTEX_CMP_REQUEUE_PI`). This listing omits a couple more obscure operations that are not particularly relevant to the problem at hand.

`FUTEX_WAIT` implements a wait-queue using futexes. The calling thread is suspended until it is either notified, times out, or receives a signal. A key implementation detail worth noting is that before the thread is suspended, the kernel compares the expected value of the futex (`val1`) with what it reads from the userspace address (`addr1`). If they do not match, `-EWOULDBLOCK` is returned to userspace. This check ensures that the kernel and userspace are in agreement with the state of the futex. This ensures, for instance, that a wake-up is not issued immediately before the waiting thread is enqueued on the futex and suspended, only to sit there indefinitely having missed the wake-up.

`FUTEX_WAKE` is used to wake one or more threads blocked on a futex. As an update to the "Futexes are Tricky" article [12], futex queues are now implemented with a `plist` and waiters are woken in priority order.

`FUTEX_CMP_REQUEUE` is used to wake one or more threads waiting on `addr1` and move the rest to wait on `addr2` without waking them first. As with `FUTEX_WAIT`, the expected value of the first futex is compared prior to performing the requeue; `-EAGAIN` is returned in the event of a mismatch. This protects against another wait, wake, or requeue event completing before the hash-list locks can be acquired. Once acquired, the futex value is guaranteed not to change by convention, as userspace obviously could write to the value at any time. The obvious user of this operation is `pthread_cond_broadcast()`.

The PI futex operations diverge from the others in that they impose a policy describing how the futex value is to be used. If the lock is unowned, the futex value shall be 0. If owned, it shall be the thread id (tid) of the owning thread. If there are threads contending for the lock, then the `FUTEX_WAITERS` flag is set. With this policy in place, userspace can atomically acquire an unowned lock or release an uncontended lock using an atomic instruction and their own tid. A non-zero futex value will force waiters into the kernel to lock. The `FUTEX_WAITERS` flag forces the owner into the kernel to unlock. If the callers are forced into the kernel, they then deal directly with an underlying `rt_mutex` which implements the priority inheritance semantics. After the `rt_mutex` is acquired, the futex value is updated accordingly, before the calling thread returns to userspace.

For an exhaustive discussion of the various operations and the subtleties of the interface, see [12], [3], [1], and of course, `kernel/futex.c`.

Glibc uses futexes to implement both pthread condition variables and pthread mutexes, which are the two primitives relevant to this discussion. Pthread barriers are also implemented using futexes. Other locking mechanisms, such as System V semaphores [9], are unrelated to futexes.

4 The Solution

The core of the problem is ensuring that an `rt_mutex`, and by extension the PI futex, is never left in a state with waiters and no owner. This is a unique situation to `futex_requeue()` because the waiting task blocked on a non-pi futex (typically the wait-

queue of a condvar), and so did not prepare the necessary accounting prior to going to sleep. Therefore, before it can wake up, this accounting must be done on its behalf by another thread - proxy locking as I refer to it in the source. Figure 5 illustrates the desired result.

With the kernel handling the locking of the associated mutex, glibc must be updated to not try to acquire the lock after wake-up in the PI case. Since there is no POSIX interface to explicitly set the type of a pthread condvar, the condvar will need to be able to dynamically determine if it should use a PI-aware mutex for its internal lock based on the mutex the user associates with it at wait time.

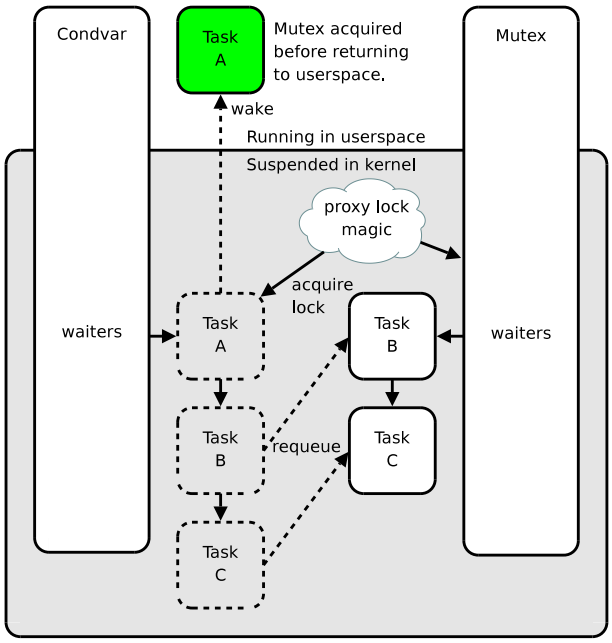

FIGURE 5: *Thread state after pthread_cond_broadcast() (PI case with requeue PI support).*

4.1 Kernel Details

4.1.1 RT-Mutex Proxy Locking

As the solution revolves around the `rt_mutex`, a basic familiarity with its usage is necessary. Acquiring an `rt_mutex` is always done in two basic steps: try to acquire it atomically, failing that, block waiting for the owner to release the lock. This is actually performed in a loop as small race windows exist between the task waking up and taking the necessary locks and actually acquiring the `rt_mutex`, during which a lock steal may occur. Before blocking, the contending thread must set up an `rt_mutex_waiter` and add it to the `rt_mutex.wait_list`. It is from this list that the next owner is selected when the `rt_mutex` is released. Some other accounting is also performed related to maintaining the PI chain to ensure the owner is always running at the priority of the highest priority waiter. In order to implement userspace-accessible-PI-aware locking primitives, futexes are bound to an `rt_mutex` via the `futex_pi_state` structure. The details of the priority inheritance implementation and the priority boosting mechanism are beyond the scope of of this paper, refer to [1] and [15] for more details.

Requeuing a futex involves two separate execution contexts: the waiter and the signaler, corresponding to `pthread_cond_wait()` and `pthread_cond_broadcast()` respectively. The waiter will suspend on a futex and the signaler will wake or requeue it to another futex. As the intended result is serialized priority ordered wake-up, to avoid the thundering herd, the requeue will wake at most one thread and requeue the rest.

If the requeue target (corresponding to the PI mutex associated with the condvar) is held for the duration of the requeue, then no wake-up will occur. However, a simple requeue is not sufficient as the requeued threads are now officially waiters for the `rt_mutex` backing the requeue target futex. The `rt_mutex_waiter` structures must be set up and the various PI chain structures must be updated so that, if necessary, the priority of the `rt_mutex` owner may be boosted to that of the highest priority requeued task (the top-waiter). When the owner releases the lock, the top-waiter will be woken and will need to complete the lock acquisition the signaler started on its behalf.

Should the requeue target be uncontended, or if a lock steal is possible [7], the top-waiter should acquire the `rt_mutex` and return to userspace. Unfortunately, the lock acquisition cannot be left up to the waiter for reasons discussed earlier (a race back to userspace that may leave the `rt_mutex` with waiters and no owner). The signaler must acquire the `rt_mutex` on behalf of the top-waiter prior to either of them returning to userspace.

Two new `rt_mutex` routines were added to enable proxy locking: `rt_mutex_start_proxy_lock()` and `rt_mutex_finish_proxy_lock()`. First, `rt_mutex_slowlock()` was carefully refactored into its atomic and blocking sections and assumptions about the task to acquire the lock being `current` were removed. This refactoring maximizes the amount of

[7] It is also possible that the owning task died and we are using robust futex.

code reused in the new routines.

The proxy lock process begins with a call to `rt_mutex_start_proxy_lock()`, as defined in Listing 5.

```
int rt_mutex_start_proxy_lock(
  struct rt_mutex *lock,
  struct rt_mutex_waiter *waiter,
  struct task_struct *task,
  int detect_deadlock)
```

Listing 5: rt_mutex_start_proxy_lock() definition.

Here, `task` is the task for which `lock` is to be acquired. After taking the `lock.wait_lock`, an atomic acquisition is attempted which will succeed if the lock is uncontended (no owner and no waiters) or if a lock steal succeeds. If successful, `task` is set as the new owner, and the routine returns. Failing an atomic lock, the `waiter` structure is prepared and the PI chain is propagated via `task_blocks_on_rt_mutex()`, but the current execution context is not suspended since it is not acquiring the lock for itself. `task` is now blocked on `lock` as though it had called `rt_mutex_lock()` itself.

If the lock was not acquired atomically, the waiter will have to complete the lock acquisition when it is woken with a call to `rt_mutes_finish_proxy_lock()`, as defined in Listing 6.

```
int rt_mutex_finish_proxy_lock(
  struct rt_mutex *lock,
  struct hrtimer_sleeper *to,
  struct rt_mutex_waiter *waiter,
  int detect_deadlock)
```

Listing 6: rt_mutex_finish_proxy_lock() definition.

With the `waiter` already prepared and the PI chain accounting taken care of, the newly woken waiter reuses the refactored `__rt_mutex_slowlock()` to complete the lock acquisition. This will try to take the lock atomically, which will often succeed in the common case where the task is woken by the previous owner. Should the lock be stolen by a higher priority waiter, the task will suspend itself and wait for another wake-up. The loop ends when the lock is acquired or the timeout, `to`, expires.

4.1.2 New Futex Operations

The restrictions imposed by the use of the `rt_mutex` made it necessary to create two new futex operations: `FUTEX_WAIT_REQUEUE_PI` and `FUTEX_CMP_REQUEUE_PI`. The former demuxes through the system call to a new function, `futex_wait_requeue_pi()`, while the existing `futex_requeue()` was able to handle the latter with some modifications. The new `futex_wait_requeue_pi()` function is the result of a high-speed collision between the existing `futex_wait()` and `futex_lock_pi()`, both of which were refactored slightly in order to maximize code reuse.

A task wishing to block on a futex (futex1) until another thread indicates it should try and acquire another futex (futex2) (such as `pthread_cond_wait()`) starts by calling:

```
futex_wait_requeue_pi(uaddr, fshared,
  val, timeout, bitset, clockrt, uaddr2);
```

Here, `uaddr` is the userspace address representing futex1, `fshared` is 0 (implying a private mapping), `val` is the expected value of `uaddr`, `timeout` is the timeout in absolute time, and `bitset` is currently unused and may be removed in a future revision. `clockrt` is 1 or 0, indicating which clock id (`CLOCK_REALTIME` or `CLOCK_MONOTONIC`) `abs_time` is represented in. Finally, `uaddr2` is the userspace address representing futex2.

`futex_wait_requeue_pi()` will first go through the same steps as `futex_wait()` to prepare to block on futex1, but prior to doing so will also prepare an `rt_mutex_waiter` and a `futex_key` (encoding futex2) on the stack and reference then in the `futex_q` for the requeue code to reference later. Once suspended on futex1, the waiter will wake due to a futex wake-up event, a signal, or timeout, in one of two states: while queued on futex1 or after requeue to futex2. By way of introduction, consider the expected path: being deliberately woken due to a futex action (rather than a signal or a timeout) after being requeued to futex2. Once woken, the waiter must determine if the `rt_mutex` has been fully acquired on its behalf, by `futex_requeue()`, or if it must complete the lock acquisition. This is a simple matter of testing the `futex_q.rt_waiter` for `NULL`, as shown in the overly simplified pseudo-code in Listing 7.

```
if (!q.rt_waiter)
  return 0;
else
  rt_mutex_finish_proxy_lock(pi_mutex,
    to, &rt_waiter, 1);
```

Listing 7: Simplified pseudo-code of the lock acquisition test in futex_wait_requeue_pi().

If the lock was acquired atomically by the requeue code, the `rt_waiter` in the `futex_q` will be NULL. If it is not, the waiter must now complete the lock acquisition via the proxy locking code described earlier. Either way, the waiter will return to userspace holding futex2 and the `rt_mutex` associated with it.

Due to the nature of futexes, which necessitates access to userspace values from within the kernel, the actual implementation is much less cut and dry. Futex code is susceptible to faults, signals, timeouts, and must protect the kernel from any sort of evil userspace can conceive to throw through the system call. These routines were carefully constructed to group code that may fault together and place it early in the routine to minimize cleanup should a fault occur.

The new `futex_wait_requeue_pi()` function presents a particular challenge due to its numerous potential states. Figure 6 illustrates a slightly simplified state transition diagram for the life-cycle of the function. In the PRE WAIT state, the function checks its arguments and read-write permission on the futex address (where it may fault), prepares the `futex_q`, sets up the timeout (if there is one), and acquires the hash-list lock. If the bitset is empty or if an unrecoverable fault occurs, the routine returns an appropriate error code. Once the hash-list lock is acquired, the routine enters the QUEUEING state where it will compare the expected futex value with the actual value (where it may have to drop the hash-list lock and return to PRE WAIT to handle a fault, or return `-EFAULT` in the event of an unrecoverable fault), if the values differ, it returns `-EWOULDBLOCK`. If they match, the task is suspended on the first futex and enters the QUEUED state. This is the first "sleep state", indicated in gray.

From any sleep state, a futex routine will resume execution for one of three reasons: a signal was delivered, a timeout occurred, or a futex wake event occurred. Each of these must be processed on exit from the QUEUED state. Before exiting with `-ETIMEDOUT` or `-ERESTARTNOINTR` (due to a signal), the QUEUED state must first determine which futex the task was woken on. By comparing the `futex_q.key` with that of the second futex, it determines if the task was requeued by `futex_requeue()` already. If the task was woken prior to requeue, then the cause of the wakeup is either a timeout or a signal. The appropriate error code is returned. If the task was requeued prior to wakeup, then those are still valid causes, but `futex_requeue()` may have also acquired the lock on its behalf, in which case the signal and timeout are ignored, and the routine returns 0, indicating a successful wakeup and lock acquisition. This can be determined by testing the `futex_q.rt_waiter` which `futex_requeue()` will set to NULL in the event of an atomic lock acquisition. If the `rt_waiter` is non-null, the REQUEUED state is entered, meaning the task has been requeued, but the lock has not yet been acquired.

At this point, before checking for a timeout or signal, the routine attempts to atomically acquire the lock in case it became available. Failing that, it will check for timeout and signal. In the event of a timeout, `-ETIMEDOUT` is returned. If a signal is pending, the system call could be restarted, but since the requeue has already occurred the futex value will have changed, causing the restarted syscall [10] to immediately return `-EWOULDBLOCK`. As such, in the event of a signal `-EWOULDBLOCK` is returned as a slight optimization. Finally, if no timeout or signal was received, the task will continue to finalize the `rt_mutex` acquisition. If contended, or in the event of a lock steal, the task is suspended. It will wake due to a signal, timeout, or `rt_mutex` unlock event, at which point the test for lock acquisition, signal, and timeout continues until one of the three is true.

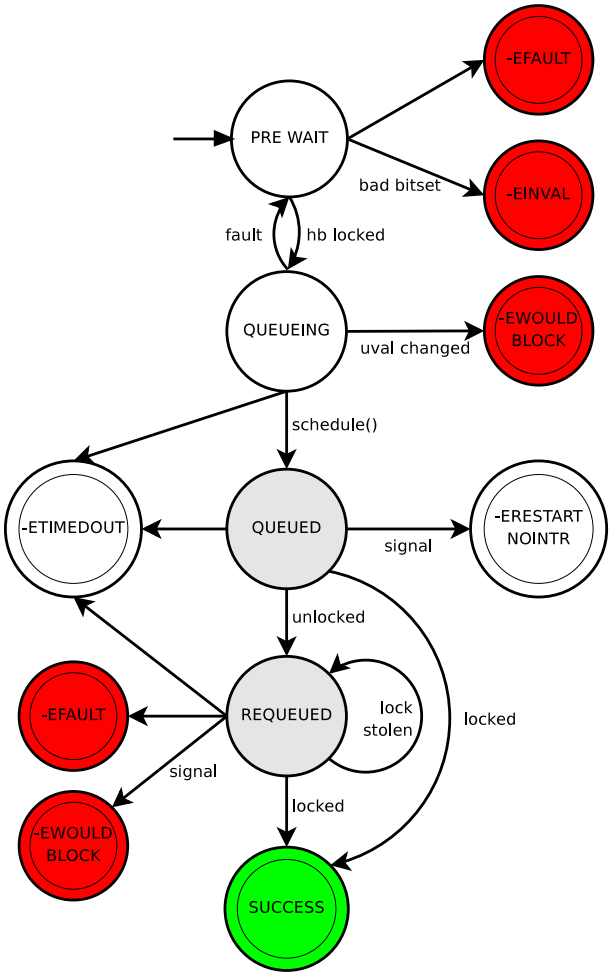

FIGURE 6: *State transition diagram for futex_wait_requeue_pi().*

Fortunately, `futex_requeue()` was able to be modified to support requeue PI and a new function was not needed. A task wishing to wake the

tasks blocked on futex1 and have them wake and acquire futex2 in priority order (such as `pthread_cond_broadcast()`) begins by calling:

```
int futex_requeue(uaddr1, fshared,
  uaddr2, nr_wake, nr_requeue, cmpval,
  requeue_pi)
```

Here, `uaddr1` is the user address representing futex1, `fshared` is 0 (indicating a private mapping), `uaddr2` is the user address representing futex2, `nr_wake` is 1 (any other value is invalid as only one task can own the `rt_mutex` at any given time), `nr_requeue` indicates the number of tasks to requeue (usually 0 or `INT_MAX` for all), `cmpval` is the expected value of futex1, and `requeue_pi` is 1 (indicating futex2 is PI futex).

Figure 7 shows the state diagram for `futex_requeue()` with the `requeue_pi` variable set to 1. In the SETUP state, the function checks for valid arguments, ensures a `pi_state` has been allocated for use, and ensures appropriate read-write permissions on the two futexes. If memory cannot be allocated for the `pi_state`, `-ENOMEM` is returned. Once complete, both hash-list locks are acquired, and the function enters the ATOMIC LOCK state.

Here, the function compares the expected value with the actual value of futex1 (which may fault and force an unlock of the hash-list locks and a return to the SETUP state). If the values do not match, `-EAGAIN` is returned. Next, the routine attempts to acquire the lock on behalf of the top-waiter. If the lock cannot be acquired, or if `nr_requeue` is non-zero, the `FUTEX_WAITERS` bit is set for futex2. This may result in a fault, which is easier to deal with before entering the REQUEUE LOOP state. If the lock succeeds, as it will if the lock is uncontended or if a lock steal occurs, the top-waiter's `futex_q` is removed from its hash-list, its `futex_q.key` is updated to that of futex2 (to indicate a requeue), its `futex_q.lock_ptr` is updated to that of the hash-list for futex2, and its `futex_q.rt_waiter` is set to NULL to indicate the `rt_mutex` has been acquired on its behalf. The top-waiter is then woken, at which point it will resume from the QUEUED state in `futex_wait_requeue_pi()` and promptly return to userspace with the lock held. If this atomic lock acquisition fails, the top-waiter will simply be requeued with the remaining `nr_requeue` waiters in the next state. For the first waiter on the `rt_mutex`, the previously allocated `pi_state` is setup with the current owner of futex2 as the new `rt_mutex` owner (this may occur here or in the REQUEUE LOOP state). If, however, the `futex_q.requeue_pi_key` is found not to match `key2`, then `-EINVAL` will be returned to userspace, indicating that userspace tried to requeue to a target futex other than that which the waiter was expecting.

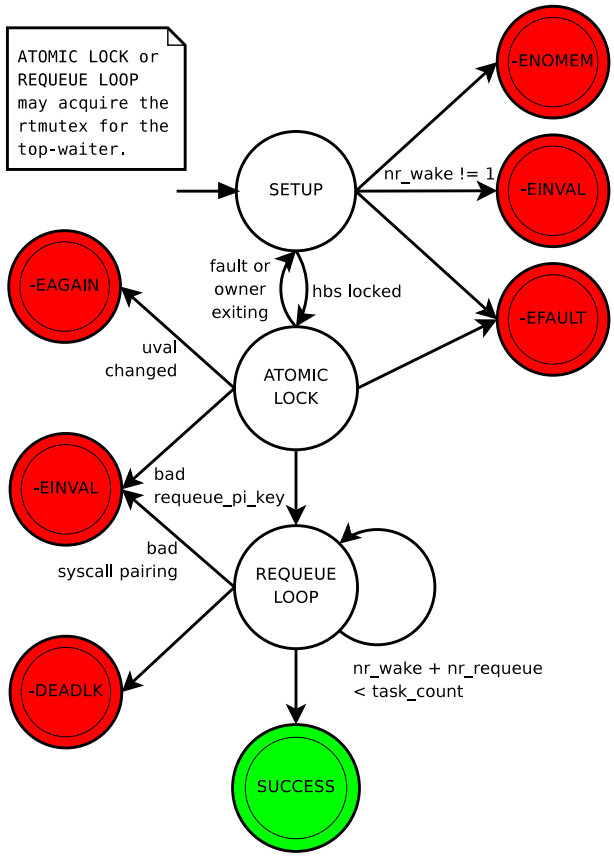

FIGURE 7: *State transition diagram for futex_requeue().*

The REQUEUE LOOP state follows and is the core of the `futex_requeue()` routine. For each matching `futex_q` on the first hash-list, the loop will assign the `futex_q.pi_state` to the that representing the `rt_mutex` backing futex2 (described above) and call `rt_mutex_start_proxy_lock()`. This call will return 1 if the lock was acquired atomically, 0 on a successful enqueueing on the `rt_mutex`, and `-EDEADLK` for various deadlock scenarios. On an atomic lock acquisition, the task is woken in same manner as in ATOMIC LOCK. On a successful enqueue of the task on the `rt_mutex`, the `futex_q` is removed from the first hash-list and placed on the second while its `lock_ptr` is updated accordingly. Once the loop has completed, the total number of tasks woken or requeued is returned to indicate success. The requeue loop may be interrupted in a few special cases. If `requeue_pi` is 1 and the current `futex_q` (`this`) has a NULL `rt_waiter`, `-EINVAL` is returned, indicating that the user mismatched a requeue-PI call with a non-requeue-PI call. The two

new futex operations, FUTEX_WAIT_REQUEUE_PI and FUTEX_CMP_REQUEUE_PI must be paired only with each other. As with the earlier atomic lock attempt, if the futex_q.requeue_pi_key does not match key2, -EINVAL is returned. Finally, in the event of a deadlock, -EDEADLK is propagated to userspace.

4.2 Glibc Details

4.2.1 Making Use of Requeue PI

The changes needed to the glibc implementation of the pthread_cond* functions to make use of requeue PI are straight forward. Prior to issuing the system call, they now check the mutex associated with the condvar to see if the PI attribute has been set (PTHREAD_PRIO_INHERIT). If so, pthread_cond_wait() will use FUTEX_WAIT_REQUEUE_PI, rather than FUTEX_WAKE for all the waiters as was done previously, as shown in line 9 of Listing 8. The reader will recall that the mutex has been acquired in the kernel upon a successful return from FUTEX_WAIT_REQUEUE_PI. Therefore, the updated implementation skips the call to lock the mutex in the PI case, as shown in lines 16-17 of Listing 8.

```
1  cond_wait(cond, mutex)
2  {
3       lock(cond->__data.__lock);
4       unlock(mutex);
5       do {
6            unlock(cond->__data.__lock);
7            if (mutex->kind == PI)
8              futex(cond->__data.__futex,
9                    FUTEX_WAIT_REQUEUE_PI);
10           else
11             futex(cond->__data.__futex,
12                   FUTEX_WAIT);
13           lock(cond->__data.__lock);
14      } while(...)
15      unlock(cond->__data.__lock);
16      if (mutex->kind != PI)
17        lock(mutex);
18 }
```

Listing 8: PI-aware pthread_cond_wait pseudo-code.

Similarly, pthread_cond_broadcast() will use FUTEX_CMP_REQUEUE_PI, as opposed to FUTEX_CMP_REQUEUE, as shown in lines 5-10 of Listing 9.

With these changes in place, user applications using PI mutexes with condvars can now benefit from the kernel implementation of requeue PI. The mutex will be acquired in priority order upon signaling the waiters of a condvar and scheduling overhead will be reduced.

```
1  cond_broadcast(cond, mutex)
2  {
3       lock(cond->__data.__lock);
4       unlock(cond->__data.__lock);
5       if (mutex->kind == PI)
6         futex(cond->__data.__futex,
7               FUTEX_CMP_REQUEUE_PI);
8       else
9         futex(cond->__data.__futex,
10              FUTEX_CMP_REQUEUE);
11 }
```

Listing 9: PI-aware pthread_cond_broadcast pseudo-code.

4.2.2 PI-Aware Condvars

Making use of the new futex operations is just the first part of making condition variables fully PI-aware. The internal lock protecting the condvar's state is a simple futex, and therefore not PI-aware. As stated in Section 1, this can lead to an unbounded priority inversion. Consider a scenario where a low priority task acquires the internal condvar lock as part of a pthread_cond* operation but is preempted before it can release the lock. Any high priority task that attempts an operation on the same condvar will now be subjected to an unbounded priority inversion [5]. The fix is to use the PI futex APIs for handling the internal lock if the associated mutex has the PI attribute set, as shown in Listings 10 and 11.

```
1  lock(cond)
2  {
3       mutex = cond->mutex;
4       if (mutex->kind == PI)
5          futex(cond->__data.__lock,
6                FUTEX_LOCK_PI);
7       else
8          futex(cond->__data.__lock,
9                FUTEX_WAIT);
10 }
```

Listing 10: PI-aware condvar lock() pseudo-code

```
1  unlock(cond)
2  {
3       mutex = cond->mutex;
4       if (mutex->kind == PI)
5          futex(cond->__data.__lock,
6                FUTEX_UNLOCK_PI);
7       else
8          futex(cond->__data.__lock,
9                FUTEX_WAKE);
10 }
```

Listing 11: PI-aware condvar unlock() pseudo-code

Conceptually, these changes are relatively trivial. Indeed, their implementation in the C language

versions of the respective files was trivial. However, C libraries being what they are, that was only part of the solution. Glibc maintains optimized hand-written assembly versions of all of these functions. These implementations were also updated accordingly. The interested reader is referred to the glibc libc-alpha mailing list and source repositories for details [4].

5 Results

With the solution in place, the prio-wake test passes consistently, with or without the associated mutex held across the `pthread_cond_broadcast()` call, as seen in line 12 of Listing 12. This ensures deterministic priority ordered wake-up and acquisition of the associated mutex by the waiting threads.

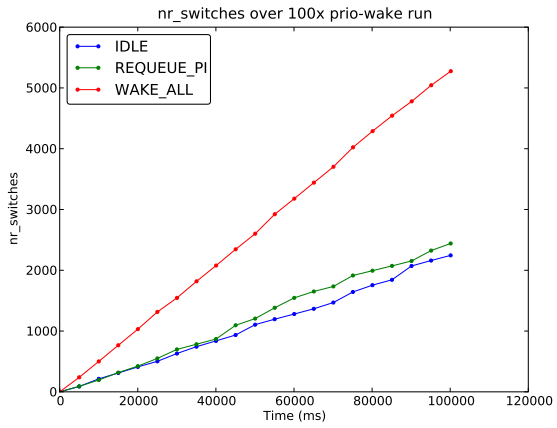

FIGURE 8: *Number of scheduler switches over time during a consecutive run of 100 prio-wake tests in which the associated mutex is not held over the broadcast. The IDLE dataset is a sampling of the system while idle for comparison.*

In addition to functional correctness, the ordered wake-up and lock acquisition also have performance benefits by eliminating the scheduling overhead the thundering herd imposes in the "wake all" approach. Figure 8 plots the number of scheduler switches [8] that occur over the course of a run of 100 consecutive prio-wake tests, with the associated mutex unlocked across the call to `pthread_cond_broadcast()`. The `WAKE_ALL` run experiences many more switches than the `REQUEUE_PI` run, which sees only a few more than the `IDLE` run. Over the course of the 102 second test, the `WAKE_ALL` run sees approximately 3,000 more switches than the `REQUEUE_PI` run, about 30 switches per second. As each iteration takes approximately one second to complete, it follows that in order to signal eight threads (one per CPU) of an event and have them acquire a shared mutex, the "wake all" approach requires 30 additional switches. The unlocked case is the more difficult one to pass using the "wake all" method as woken threads will not be hindered by the signaling thread holding the mutex, and will race only amongst themselves to return to userspace and acquire the lock, as depicted in Figure 3. Over the 100 runs, the `WAKE_ALL` run experienced 100% failure as compared to 0% for the `REQUEUE_PI` run [9].

The kernel patches have been available in the PREEMPT_RT [7] tree since as early as 2.6.29.1-rt5 and is currently destined for the upcoming mainline 2.6.31 release. At the time of this writing, the x86_64 version of the new futex operations patch is present in glibc CVS. The remaining glibc patches are still under development, and should be available in the near future.

The system used for all test results reported here was an 8-way x86_64 system. The results were collected on a Linux 2.6.31-rc6-rt4 kernel.

6 Acknowledgements

We would like to extend our thanks to Thomas Gleixner for his initial conceptual designs and facilitating the inclusion of the the kernel and glibc patches into their respective upstream repositories. Thanks to Peter Zijlstra for his efforts to reduce overhead related to accessing user pages in the futex code as well as his sound advice at various points throughout the development process. Will Schmidt, Paul McKenney and Venkateswararao Jujjuri (JV) provided early reviews which were critical to making this paper more legible than it would have been otherwise. Thanks to IBM for the resources, both hardware and person-hours, required to complete this project. Finally, we are indebted to our families for their sacrifices to enable us to pursue this effort.

[8] As reported by the `/proc/sched_debug nr_switches` field.

[9] At the time of this writing, the performance benefits are not observed with the associated mutex held over the pthread_cond_broadcast() call. This is considered to be a flaw and is expected to be resolved as soon as possible. The functional results remain intact however.

```
# LD_LIBRARY_PATH=/test/dvhart/lib/x86_64 ./prio-wake

-------------------------
Priority Ordered Wakeup
-------------------------
Worker Threads: 8
Calling pthread_cond_broadcast() with mutex: LOCKED

00000620 us: Master thread about to wake the workers

Criteria: Threads should be woken up in priority order
Result: PASS
000000: 00000193 us: RealtimeThread-17617056 pri 001 started
000001: 00000275 us: RealtimeThread-17617552 pri 002 started
000002: 00000323 us: RealtimeThread-17618048 pri 003 started
000003: 00000375 us: RealtimeThread-17618544 pri 004 started
000004: 00000427 us: RealtimeThread-17619040 pri 005 started
000005: 00000479 us: RealtimeThread-17619536 pri 006 started
000006: 00000526 us: RealtimeThread-17620032 pri 007 started
000007: 00000575 us: RealtimeThread-17620528 pri 008 started
000008: 00000666 us: RealtimeThread-17620528 pri 008 awake
000009: 00000679 us: RealtimeThread-17620032 pri 007 awake
000010: 00000689 us: RealtimeThread-17619536 pri 006 awake
000011: 00000706 us: RealtimeThread-17619040 pri 005 awake
000012: 00000716 us: RealtimeThread-17618544 pri 004 awake
000013: 00000725 us: RealtimeThread-17618048 pri 003 awake
000014: 00000735 us: RealtimeThread-17617552 pri 002 awake
000015: 00000745 us: RealtimeThread-17617056 pri 001 awake
```

Listing 12: Passing prio-wake example.

References

[1] *Documentation/pi-futex.txt*. Linux kernel documentation, 2.6.31-rc6-rt4.

[2] *Documentation/rt-mutex.txt*. Linux kernel documentation, 2.6.31-rc6-rt4.

[3] *futex - Fast Userspace Locking system call*. Linux futex(2) man page, Ubuntu 9.04.

[4] Glibc resources. http://www.gnu.org/software/libc/resources.html.

[5] Internal lock in pthread struct is vulnerable to priority inversion. http://sourceware.org/bugzilla/show_bug.cgi?id=5192.

[6] Linux test project. http://ltp.sourceforge.net/.

[7] Real-time linux wiki. http://rt.wiki.kernel.org.

[8] Real time specification for java. http://www.rtsj.org.

[9] *semop, semtimedop - semaphore operations*. Linux semop(2) man page, Ubuntu 9.04.

[10] Bovet, D., and Cesati, M. *Understanding the Linux Kernel 3rd Edition*, 3 ed. O'Reilly Media, Inc., November 2005.

[11] Butenhof, D. *Programming with POSIX Threads*. Addison-Wesley, Boston, MA, USA, 1997.

[12] Drepper, U. Futexes are tricky. http://people.redhat.com/drepper/futex.pdf, August 2009.

[13] Franke, H., Russell, R., and Kirkwood, M. Fuss, futexes and furwocks: Fast userlevel locking in linux. In *Ottawa Linux symposium* (2002), pp. 479–789.

[14] Peiffer, P. [patch 2.6.19.1-rt15][rfc] - futex_requeue_pi implementation (requeue from futex1 to pi-futex2). http://lkml.org/lkml/2007/1/3/62, 2007.

[15] Rostedt, S. *Documentation/rt-mutex-design.txt*, 2006. Linux kernel documentation, 2.6.31-rc6-rt4.