Requeue PI: Making Glibc Condvars PI Aware

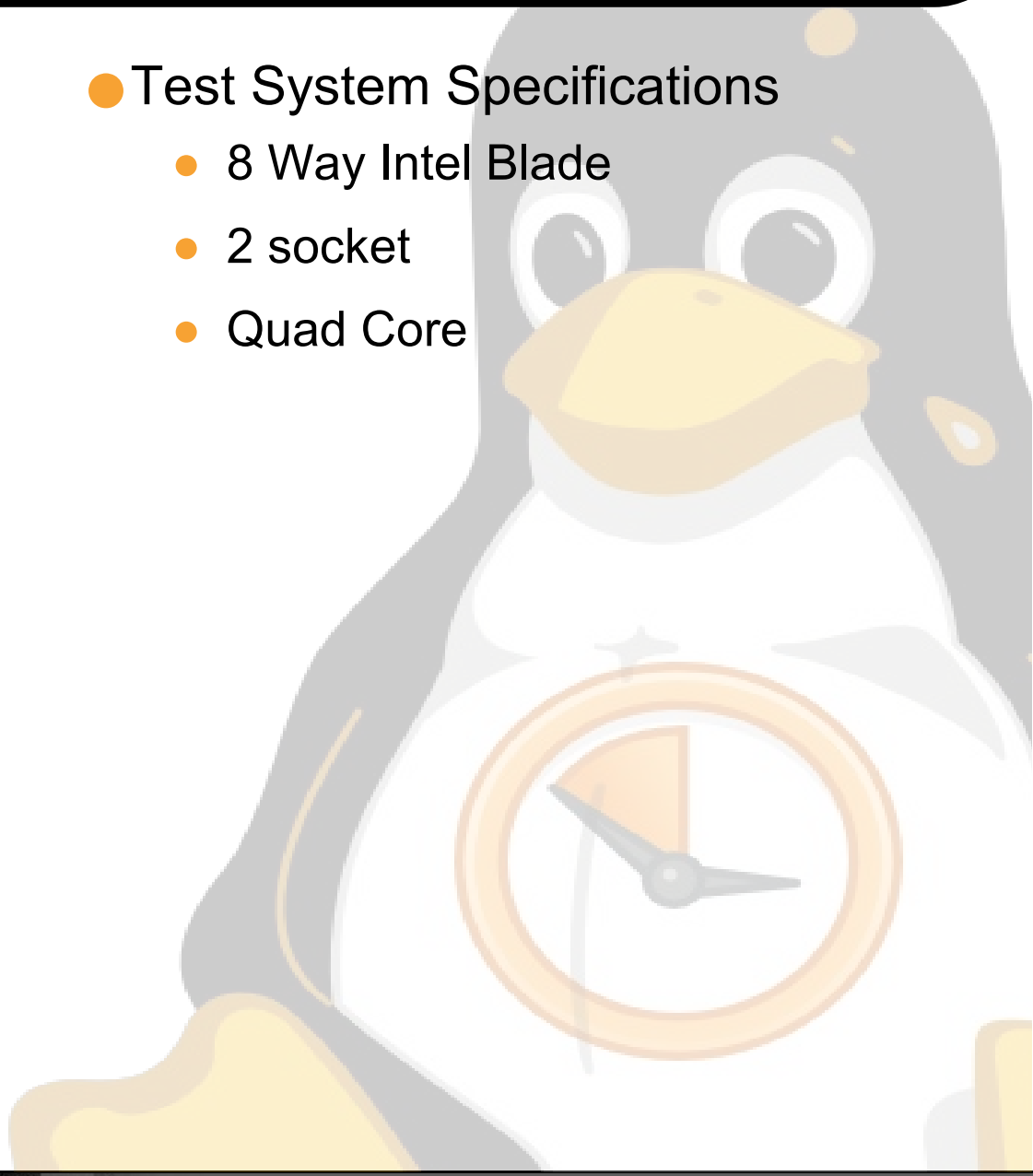
Darren Hart
Special Thanks: Thomas Gleixner and Dinakar Guniguntala

IBM Linux Technology Center
Real-Time Linux

© 2009 IBM Corporation

Agenda

- Introduction
  - Condvars
  - Futexes
- Disorderly Wakeup
  - Kernel + Glibc Solutions
- Priority Inversion
  - Glibc Solution
- Results
- Test System Specifications
  - 8 Way Intel Blade
  - 2 socket
  - Quad Core



© 2009 IBM Corporation

Introduction: Condvars

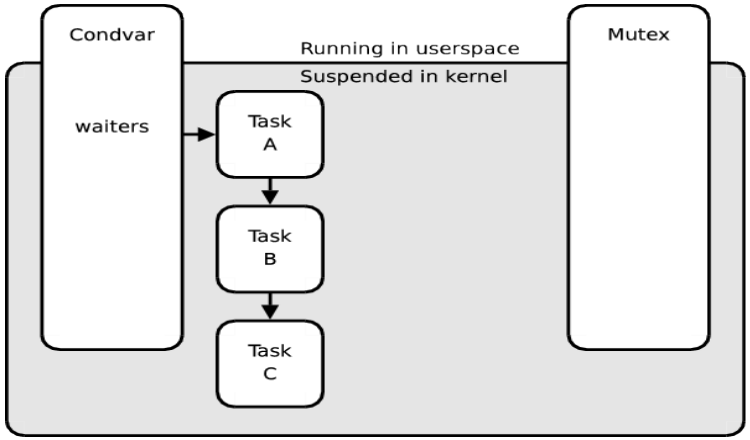
- `pthread_condvar_wait()` allows multiple threads to block on an event
- `pthread_cond_broadcast()` triggers the event

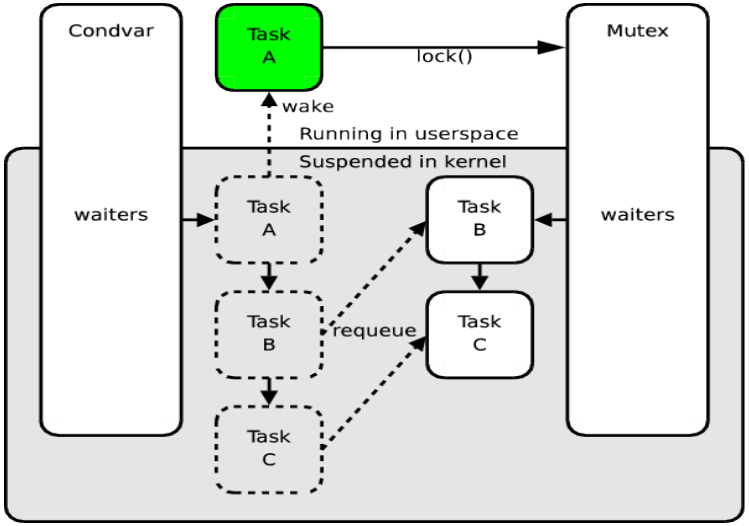
WAITING THREAD (N)

```
pthread_mutex_lock(&m);
pthread_cond_wait(&c, &m);
/*
 * block in kernel
 *
 */
do_critical_section();
pthread_mutex_unlock(&m);
```

BROADCAST THREAD (1)

```
pthread_mutex_lock(&m);
pthread_cond_broadcast(&c, &m);
/* notify the waiting threads */
pthread_mutex_unlock(&m);
```

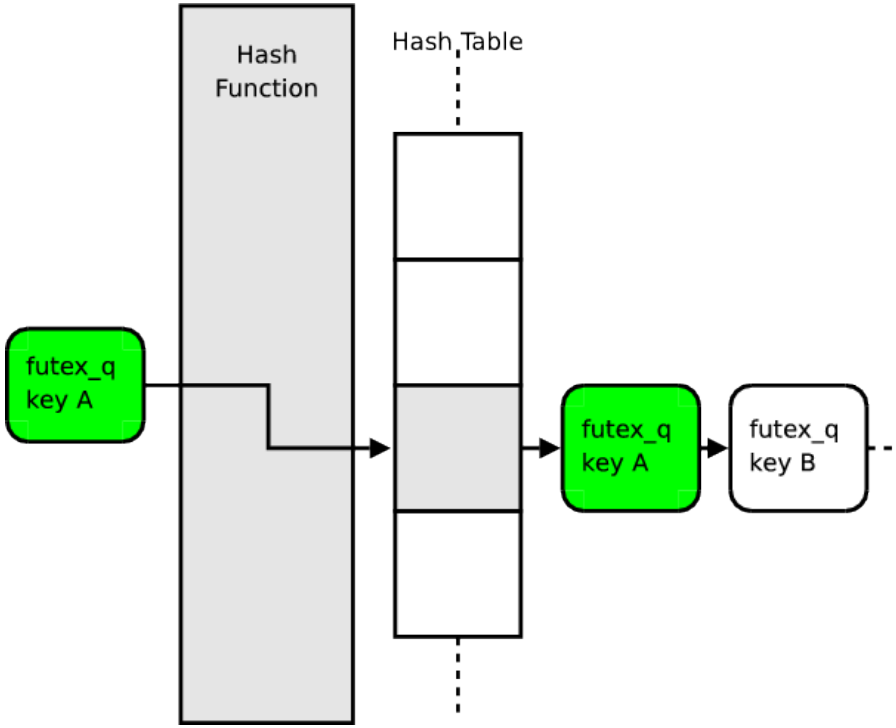




© 2009 IBM Corporation

Introduction: Futexes

- "Fast Userspace Mutexes"
- Primitive locking construct
- Userspace address
- Avoids syscall when uncontended
- Shared hashtable



© 2009 IBM Corporation

Introduction: Futexes: Operations

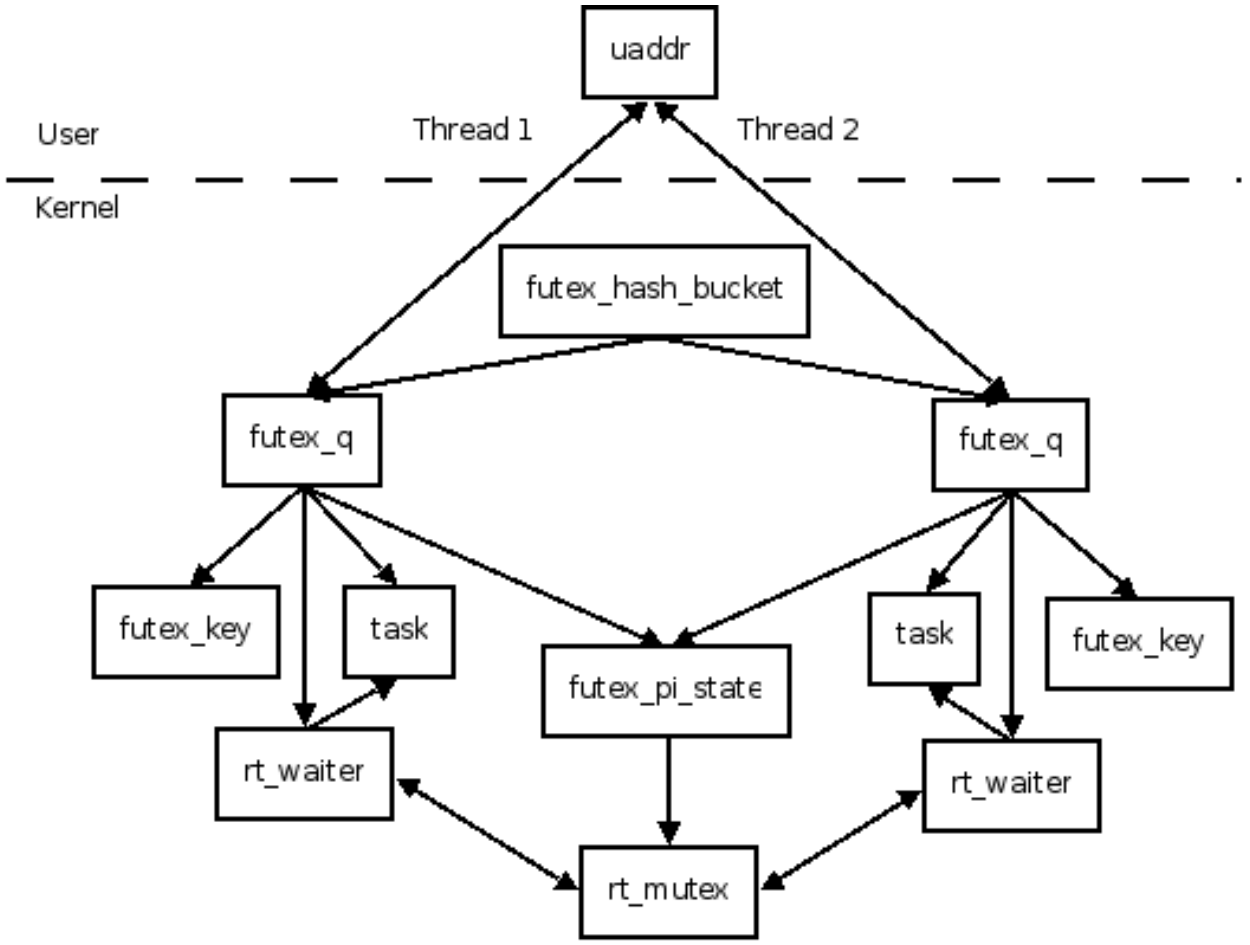
- **FUTEX_WAIT:** Block in the kernel with an optional timeout
- **FUTEX_WAKE:** Wake one or more waiters
- **FUTEX_CMP_REQUEUE:** Wake or requeue waiters to another futex
- **FUTEX_LOCK_PI:** Take a PI mutex
- **FUTEX_UNLOCK_PI:** Release a PI mutex
  - PI Futexes are implemented using the rt_mutex.
  - No ownerless rt_mutexes
  - Imposes a policy on the futex value

- See Ulrich Drepper's “Futexes are Tricky”: http://people.redhat.com/drepper/futex.pdf
- Documentation/rt-mutex-design.txt, rt-mutex.txt, pi-futex.txt

© 2009 IBM Corporation

Introduction: There is no Futex

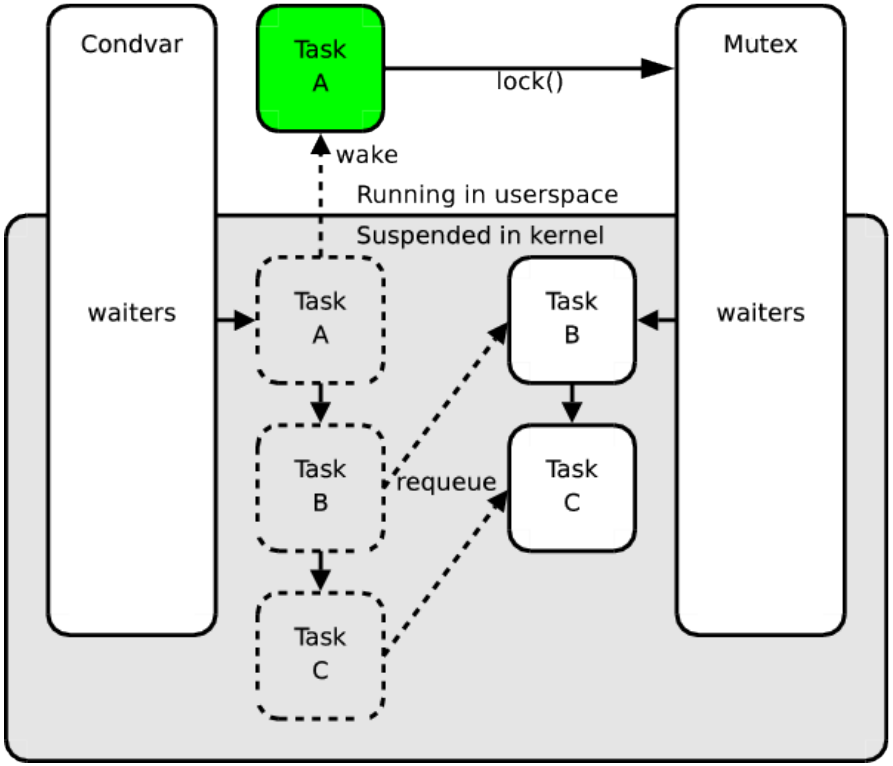
- Two threads
- Blocked on one futex
- PI Chain not shown
- Owner not shown
- Requeue PI bits not shown

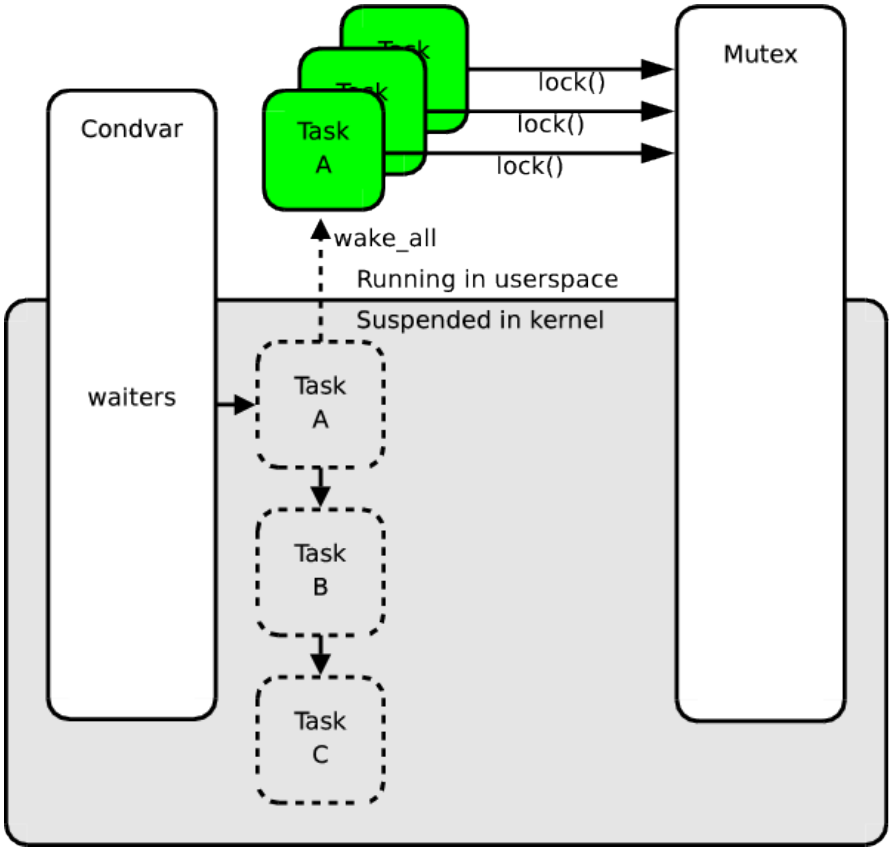


© 2009 IBM Corporation

Failure Cases: Disorderly Wakeup

- `pthread_cond_broadcast()`
  - Uses `FUTEX_CMP_REQUEUE`
  - Avoids multiple wake-up
  - Priority ordered wake-up
- No ownerless rt_mutex
- PI Mutexes use `FUTEX_WAKE(all)`
  - Wakes all waiters
  - Non-deterministic wakeup





© 2009 IBM Corporation

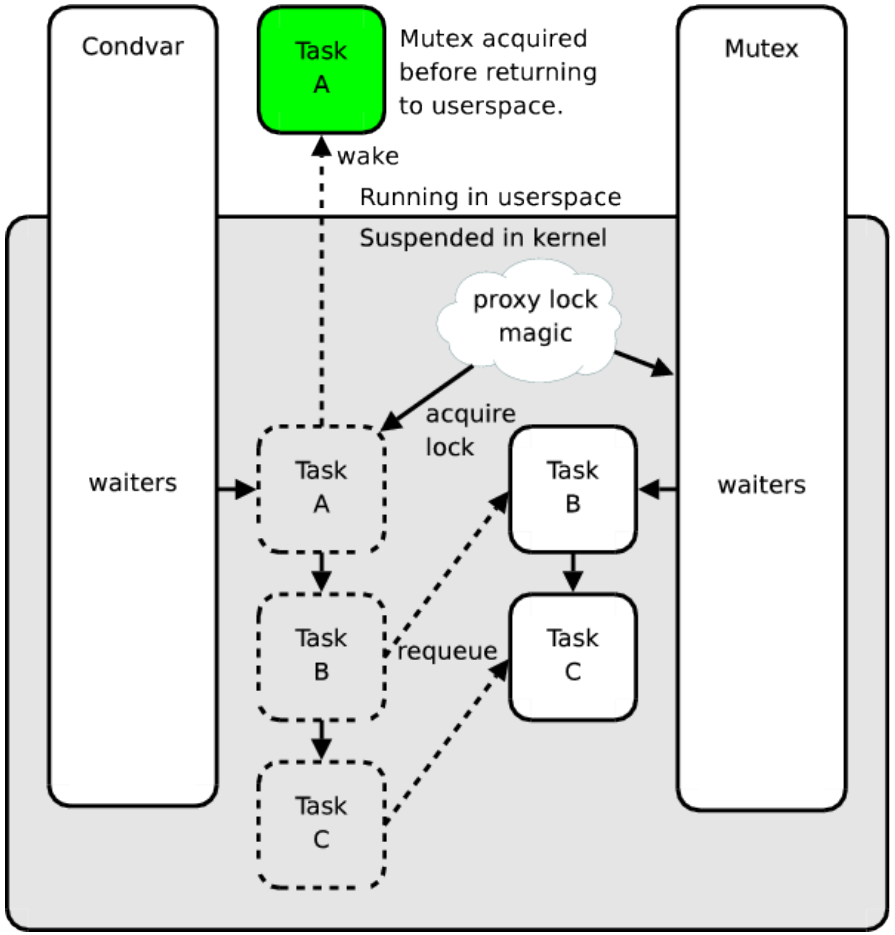
Failure Case 1: Disorderly Wakeup: Prio-Wake

```
# ./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
```


© 2009 IBM Corporation

The Solution: Overview

- Requirements
  - No ownerless rt_mutex
  - Wake only the top-waiter
  - Wake in priority order
  - Requeue from normal to PI
  - No Glibc API changes
- Challenges
  - Race back to userspace after wake
  - Must acquire lock within the kernel
- Solution
  - New futex op codes
  - New rt_mutex locking routines

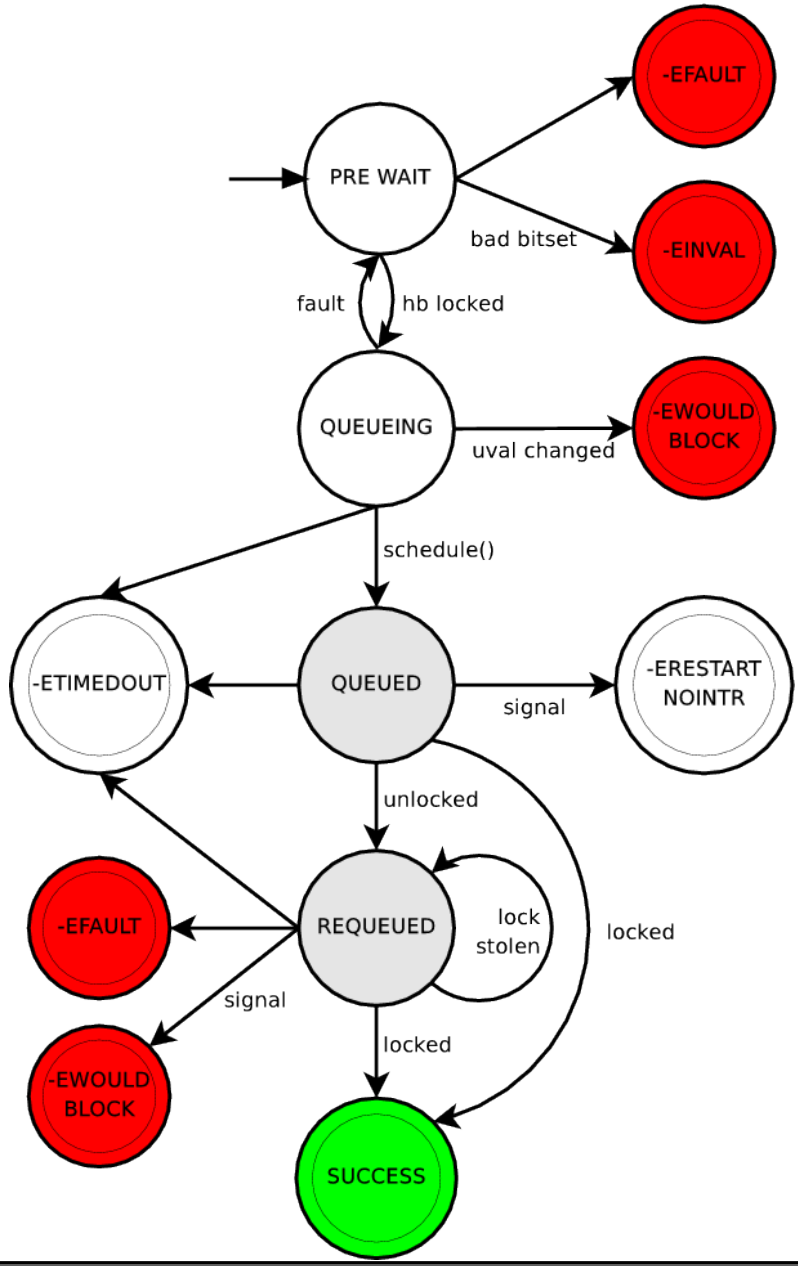


© 2009 IBM Corporation

The Solution: Kernel: FUTEX_WAIT_REQUEUE_PI

FUTEX_WAIT_REQUEUE_PI

- Used instead of `FUTEX_WAIT` in the `pthread_cond_wait()` implementation
- Specifies the target futex of the expected requeue
- Basically the result of a high-speed collision between `futex_wait()` and `futex_lock_pi()`
- Lots of abnormal exit possibilities and corner cases

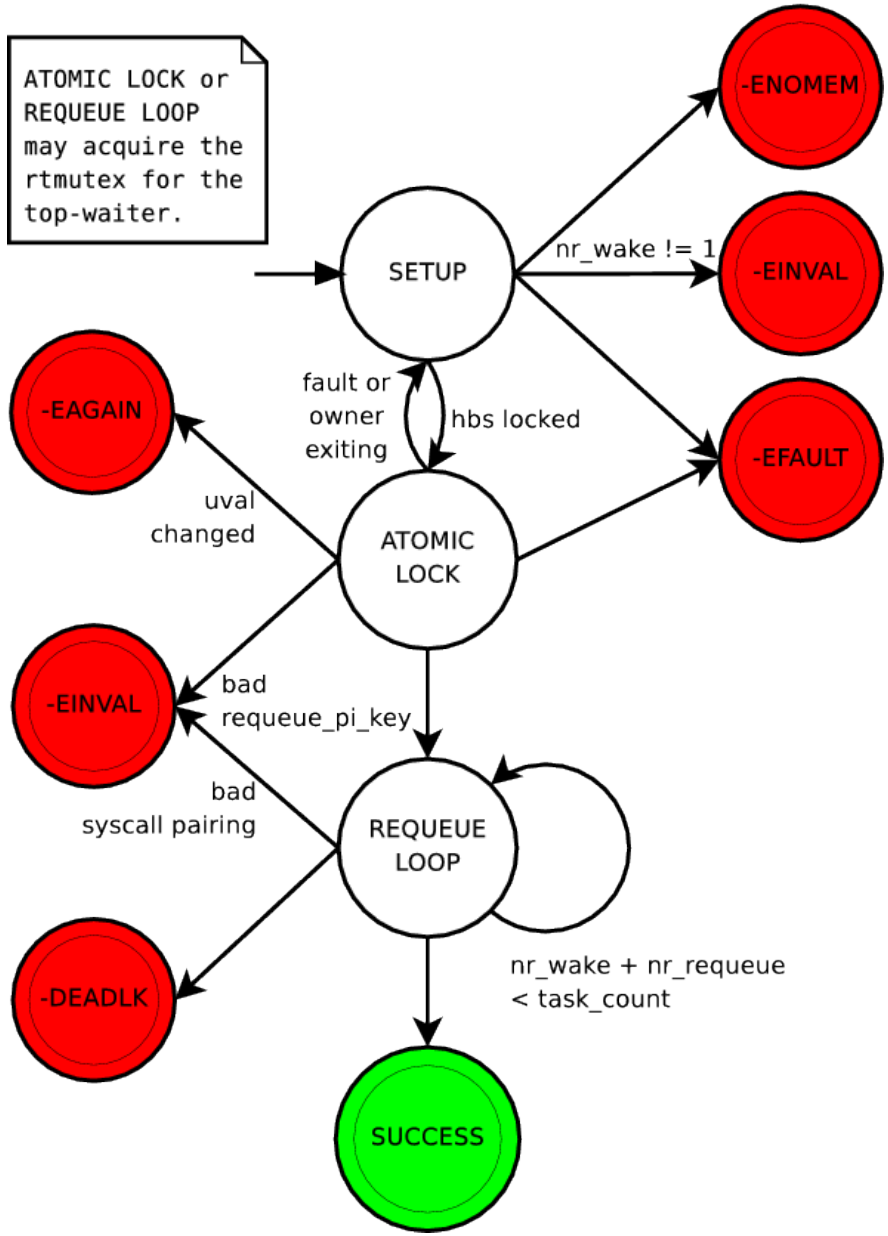


© 2009 IBM Corporation

The Solution: Kernel: FUTEX_CMP_REQUEUE_PI

FUTEX_CMP_REQUEUE_PI

- Used instead of FUTEX_CMP_REQUEUE in the `pthread_cond_broadcast()` implementation. Also used for `pthread_cond_signal()`.
- Slightly modifies the existing `futex_requeue()` function.
- Can acquire the rt_mutex on behalf of the top-waiter if it is not held at entry.



© 2009 IBM Corporation

The Solution: Kernel: Scary Bits

- Early wakeup

```
if (!match_futex(&q->key, key2)) { ... }
```

- Atomic lock acquisition

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

- Mixed target futex

```
if (!match_futex(top_waiter->requeue_pi_key, key2))
    return -EINVAL;
```

- Unqueue wakeup race

```
set_current_state(TASK_INTERRUPTIBLE);
queue_me(q, hb);
if (likely(!plist_node_empty(&q->list)))
    schedule();
```


© 2009 IBM Corporation

The Solution: Kernel: rt_mutex_start_proxy_lock()

- `rt_mutex_start_proxy_lock()`
  - Signaling thread
  - Queue the waiter on the rt_mutex
  - Don't block

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

spin_lock(&lock->wait_lock);

mark_rt_mutex_waiters(lock);

if (!rt_mutex_owner(lock) || try_to_steal_lock(lock, task)) {
    /* We got the lock for task. */
    debug_rt_mutex_lock(lock);
    rt_mutex_set_owner(lock, task, 0);
    spin_unlock(&lock->wait_lock);
    rt_mutex_deadlock_account_lock(lock, task);
    return 1;
}

ret = task_blocks_on_rt_mutex(lock, waiter, task,
   detect_deadlock);

if (ret && !waiter->task) {
   /*
    * Reset the return value. We might have
    * returned with -EDEADLK and the owner
    * released the lock while we were walking the
    * pi chain.  Let the waiter sort it out.
    */
   ret = 0;
}
spin_unlock(&lock->wait_lock);

debug_rt_mutex_print_deadlock(waiter);

return ret;
}
```


© 2009 IBM Corporation

The Solution: Kernel: rt_mutex_finish_proxy_lock()

- `rt_mutex_finish_proxy_lock`
  - Newly woken thread
  - Complete non-atomic lock acquisition

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

spin_lock(&lock->wait_lock);

set_current_state(TASK_INTERRUPTIBLE);

ret = __rt_mutex_slowlock(lock, TASK_INTERRUPTIBLE, to,
     waiter, detect_deadlock);

set_current_state(TASK_RUNNING);

if (unlikely(waiter->task))
remove_waiter(lock, waiter);

/*
 * try_to_take_rt_mutex() sets the waiter bit
 * unconditionally. We might have to fix that up.
 */
fixup_rt_mutex_waiters(lock);

spin_unlock(&lock->wait_lock);

/*
 * Readjust priority, when we did not get the lock. We might
 * have been the pending owner and boosted. Since we did not
 * take the lock, the PI boost has to go.
 */
if (unlikely(ret))
   rt_mutex_adjust_prio(current);

return ret;
}
```


© 2009 IBM Corporation

The Solution: Glibc: New Futex Operations

- Make use of the new futex operations when the mutex is PI
- `pthread_cond_*wait()`
  - use `FUTEX_WAIT_REQUEUE_PI`
  - lock already held after return from system call
- `pthread_cond_broadcast()` and `pthread_cond_signal()`
  - Use `FUTEX_CMP_REQUEUE_PI`
- All in assembly

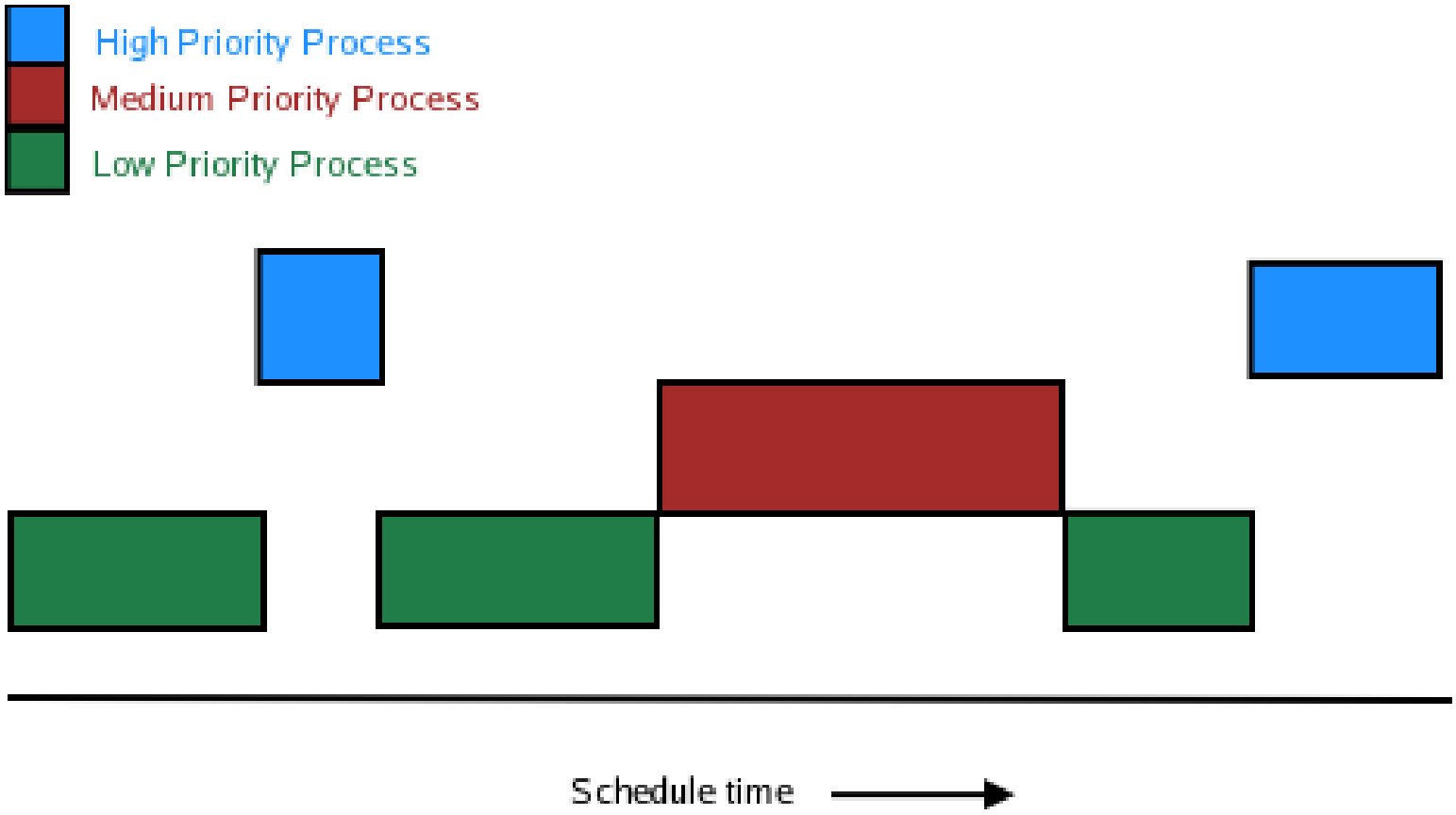
```
cond_wait(cond, mutex)
{
    lock(cond->__data.__lock);
    unlock(mutex);
    do {
        unlock(cond->__data.__lock);
        if (mutex->kind == PI)
          futex(cond->__data.__futex,
                FUTEX_WAIT_REQUEUE_PI);
        else
          futex(cond->__data.__futex,
                FUTEX_WAIT);
        lock(cond->__data.__lock);
    } while(...)
    unlock(cond->__data.__lock);
    if (mutex->kind != PI)
      lock(mutex);
}
```

```
cond_broadcast(cond, mutex)
{
    lock(cond->__data.__lock);
    unlock(cond->__data.__lock);
    if (mutex->kind == PI)
      futex(cond->__data.__futex,
            FUTEX_CMP_REQUEUE_PI);
    else
      futex(cond->__data.__futex,
            FUTEX_CMP_REQUEUE);
}
```


© 2009 IBM Corporation

Failure Case 2: Priority Inversion

- `pthread_cond_t.__data.__lock` is not a PI Mutex.
- Even if the associated mutex is!



© 2009 IBM Corporation

The Solution: Glibc: Condvar Data Lock

- Convert the internal condvar data lock to a PI mutex if the associated mutex is PI
- Use existing `FUTEX_(UN)LOCK_PI` operations

```
__lock(cond)
{
    mutex = cond->mutex;
    if (mutex->kind == PI)
       futex(cond->__data.__lock,
             FUTEX_LOCK_PI);
    else
       futex(cond->__data.__lock,
             FUTEX_WAIT);
}


__unlock(cond)
{
    mutex = cond->mutex;
    if (mutex->kind == PI)
       futex(cond->__data.__lock,
             FUTEX_UNLOCK_PI);
    else
       futex(cond->__data.__lock,
             FUTEX_WAKE);
}
```

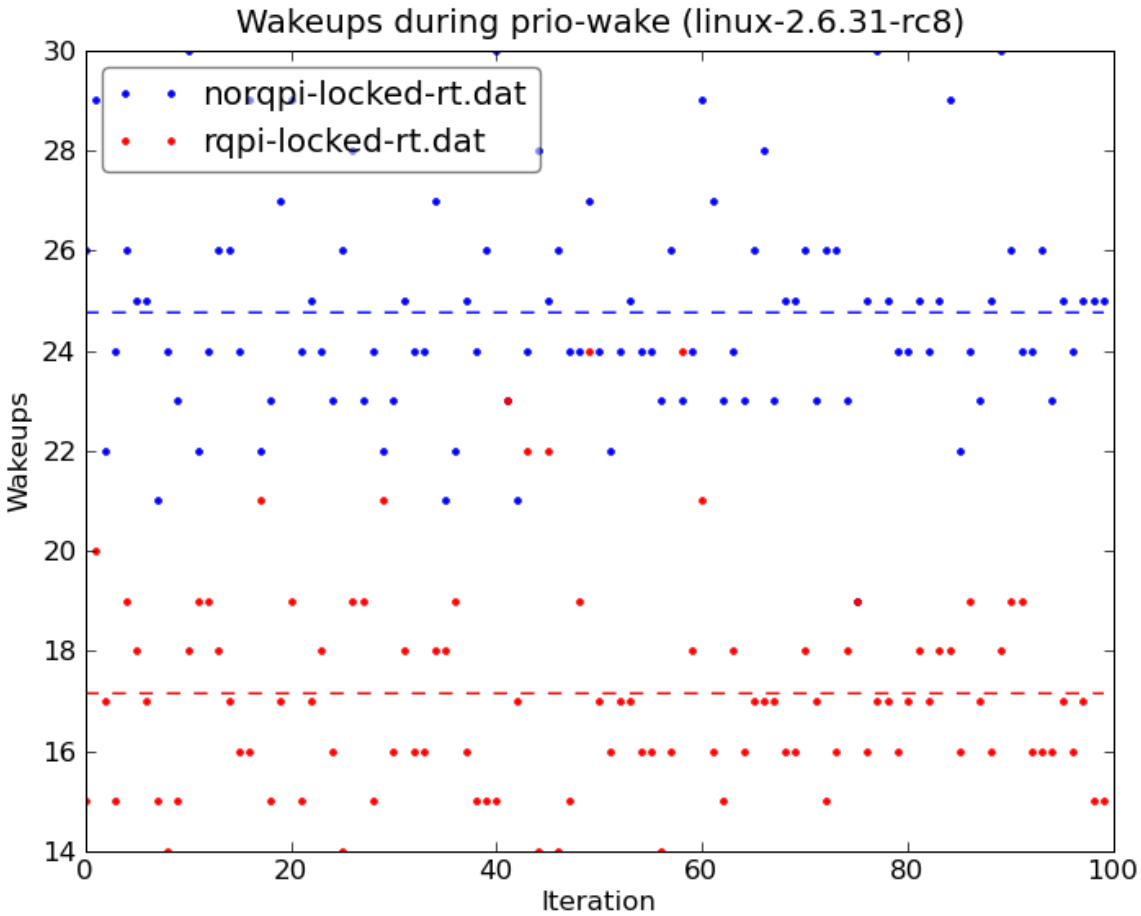

© 2009 IBM Corporation

Results: Orderly Wakeup: Prio-Wake

```
# 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
```


© 2009 IBM Corporation

Results: Orderly Wakeup: Prio-Wake (cont.)



- Wakeups measured using the ftrace sched events tracer
- Average 8 (NR_CPUS) fewer wakeup per `pthread_cond_broadcast()`
- And the test passes 100% of the time

© 2009 IBM Corporation

Future Work

- Complete futex syscall test-suite
- Consider more formal methods to detect any remaining races
- Finish merging x86 patches into Glibc
- Assist other archs to do the same

© 2009 IBM Corporation

Legal Statement

- This work represents the view of the author and does not necessarily represent the view of IBM.
- IBM, IBM (logo), e-business (logo), pSeries, e (logo) server, and xSeries are trademarks or registered trademarks of International Business Machines Corporation in the United States and/or other countries.
- Linux is a registered trademark of Linus Torvalds.
- Other company, product, and service names may be trademarks or service marks of others.

© 2009 IBM Corporation

Questions

© 2009 IBM Corporation