The path of the private futex

Embedded Linux Conference Europe 2016

Sebastian A. Siewior

Linutronix GmbH

October 12, 2016

Futex introduction

- futex.c started with Rusty Russell in v2.5.7-pre1 (March 2002).
- Two ops: FUTEX_UP, FUTEX_DOWN (later renamed to FUTEX_WAIT FUTEX_WAKE).
- Basic concept: userland tries locking first and goes to kernel if the lock is taken.

Futex introduction

```
int futex_down(struct futex *futx)
{
        if (__down(&futx->count))
                return futex(futx, -1);
        return 0;
}

int futex_up(struct futex *futx)
{
        if (__up(&futx->count))
                return futex(futx, 1);
        return 0;
}
```

It evolved

- June 2002, Async_FD (FUTEX_FD).
- May 2003 requeue (FUTEX_REQUEUE).
- May 2004 FUTEX_CMP_REQUEUE, the former has a small race.
- September 2005, FUTEX_WAKE_OP to optimize pthread_cond_signal().
- June 2006, PI FUTEX.
- May 2007 FUTEX_CMP_REQUEUE_PI.
- May 2007 private FUTEX.
- June 2007, revert FUTEX_CMP_REQUEUE_PI it is broken.
- January 2008, removal of FUTEX_FD, too racy.

FUTEX concept

- User tries to acquire a lock by the use of an atomic operation.
- If it succeeds then the kernel is not involved.
- If the lock is contended the kernel is called for help.
- The kernel serves the corner cases with little knowledge about the validity of the lock pointer.

A few details

- A struct futex_hash_bucket is obtained based on the hash of the user address pointer (lock pointer).
- Contains a spinlock list of process waiting (struct futex_q).
- The spinlock is held during queue modifications / state.
- The spinlock prevents preemption but on -RT it does not. Especially when PI is involved.

Ping pong boost on -RT

```
med   sched_wakeup:      comm=high
med   sched_switch:      prev=med/29 ==> next=high/9
high  sched_pi_setprio: comm=low oldprio=120 newprio=9
high  sched_switch:      prev=high/9 prev_state=S ==> next=low/9
low   sched_wakeup:      comm=high prio=9
low   sched_pi_setprio: comm=low oldprio=9 newprio=120

low   sched_switch:      prev=low/120 prev_state=R+ ==> next=high/9
high  sched_pi_setprio: comm=low oldprio=120 newprio=9
high  sched_switch:      prev=high/9 prev_state=D ==> next=low/9

low   sched_wakeup:      comm=high prio=9
low   sched_pi_setprio: comm=low oldprio=9 newprio=120
low   sched_switch:      prev=low/120 prev_state=R+ ==> next=high/9
high  sched_process_exit: comm=high prio=9
```

Problem identified

- !RT+SMP would spin on the lock.
- Peter Zijlstra implemented lockless wake-queues (wake_up_q()).
- Davidlohr Bueso converted futex_wake() (and ipc/mqueue) in v4.2.
- Converted futex_unlock_pi().
- ipc/msg is in akpm's queue, ipc/sem is probably a candidate.

Problem identified

- !RT+SMP would spin on the lock.
- Peter Zijlstra implemented lockless wake-queues (wake_up_q()).
- Davidlohr Bueso converted futex_wake() (and ipc/mqueue) in v4.2.
- Converted futex_unlock_pi().
- ipc/msg is in akpm's queue, ipc/sem is probably a candidate.
- The new futex_unlock_pi() broke RT due to early de-boost. Fixed in 4.6.7-rt14.

No ping pong boost on -RT

```
med  sched_wakeup:      comm=high
med  sched_switch:      prev=med/29 ==> next=high/9
high sched_pi_setprio: comm=low oldprio=120 newprio=9
high sched_switch:      prev=high/9 ==> next=low/9

low  sched_wakeup:      comm=high prio=9
low  sched_pi_setprio: comm=low oldprio=9 newprio=120
low  sched_switch:      prev=low/120 prev_state=R+ ==> next=high/9

high sched_process_exit: comm=high prio=9
```

Global hb problems

- The hb hash array is global. Not NUMA friendly.
- Two tasks can share the same bucket.
- Not always however due to ASLR.
- So it can lead to performance degradation.
- Additionally on -RT we can have unbound priority inversions. Duh!

Another hb problem

- Task A runs on CPU0 (pinned). Task B runs on CPU1.
- Task A holds the hb lock and is preempted by a task with higher priority on CPU0.
- Task B wants the hb lock but can't get it.
- Task C with a lower priority than B runs on CPU1.

v1

- Basic idea: a hb structure for every lock. More or less.
- V1 `https://lkml.kernel.org/r/20160402095108.894519835@linutronix.de`
- Opcode FUTEX_ATTACH. First create a global state (hb + futex_q).
- Keep a thread local array for lookup. Array is hashed on uaddr.
- Resize the array on collision.
- Every thread needs to attach the lock. In kernel lookup is lockless.

V1 outcome

- FUTEX_ATTACH / new ABI is something other people do not want.
- And it sounds like everyone would like this.
- Changes in glibc and kernel need time to get productive.
- Backports aren't that easy.

V1 outcome

- FUTEX_ATTACH / new ABI is something other people do not want.
- And it sounds like everyone would like this.
- Changes in glibc and kernel need time to get productive.
- Backports aren’t that easy.
- Lessons learnt:
  - “auto attach”.
  - Consider only private FUTEX.
  - Process wide. Thread wide is too complicated.

V2

- **V2** `https://lkml.kernel.org/r/20160428161742.363543816@linutronix.de`
- **Nobody cared about details. Everyone went nuts about custom hash function based on the mod function.**
- **The hash algorithm was "uaddr % prim".**

V2

- **V2** `https://lkml.kernel.org/r/20160428161742.363543816@linutronix.de`
- **Nobody cared about details. Everyone went nuts about custom hash function based on the mod function.**
- **The hash algorithm was “uaddr % prim”.**
- **How was this tested perfomance wise?**
- **perf bench futex hash -f nfutex -n node -t nthreads**
- **Performs an invalid FUTEX_WAKE over and over.**

The benchmark v2

```
23.13%  perf            [.] workerfn
23.08%  [kernel]        [k] futex_wait_setup
21.46%  [kernel]        [k] entry_SYSCALL_64_fastpath
 5.17%  [kernel]        [k] _raw_spin_lock
 4.44%  [kernel]        [k] futex_wait
 4.33%  libc-2.24.so    [.] syscall
```

The benchmark v2

```
        | for (i = 0; i < nfutexes; i++, w->ops++) {
  0.06% | bb:  mov    nfutexes,%eax
  0.00% |      add    0x1,%ebx
 93.91% |      addq   0x1,0x18(%r12)
  0.02% |      cmp    %ebx,%eax
  1.00% |      ja     68
        |                       }
        |             } while (!done);
  0.00% |      cmpb   0x0,done
        |      je     58
```

The benchmark v2

- The struct in question

```
struct worker {
        int tid;
        u_int32_t *futex;
        pthread_t thread;
        unsigned long ops;
};
```

The benchmark v2

- The struct in question

```
struct worker {
        int tid;
        u_int32_t *futex;
        pthread_t thread;
        unsigned long ops;
};
```

- How about cache line aligned?

```
struct worker {
        ....
}; __attribute__((aligned(64)));
```

The benchmark v2, take two

```
35.53%  [kernel]         [k] futex_wait_setup
11.54%  [kernel]         [k] _raw_spin_lock
 6.89%  [kernel]         [k] futex_wait
 6.70%  libc-2.24.so     [.] syscall
 6.11%  [kernel]         [k] entry_SYSCALL_64_fastpath
 6.09%  [kernel]         [k] get_futex_key_refs.isra.14
 5.41%  [kernel]         [k] hash_futex
 3.79%  [kernel]         [k] entry_SYSCALL_64
```

The benchmark v2, take two

```
       |    hash_futex():
       |       test    0x3,%al
       | struct mm_struct *mm = current->mm;
 6.27% |       mov     0x2f8(%rdx),%rcx
       |  slot = key->private.address % mm->futex_hash.hash_bits;
 0.06% |       xor     %edx,%edx
 0.11% |       mov     (%rdi),%rax
 7.50% |       mov     0x2dc(%rcx),%esi
 7.14% |       div     %rsi
       |
       |  return &mm->futex_hash.hash[slot];
 0.44% |       shl     0x6,%rdx
 1.57% |       mov     %rdx,%rax
 1.88% |       add     0x2e0(%rcx),%rax
```

LINUTRONIX
LINUX FOR INDUSTRY

The benchmark v2, take three

```
36.41%   [kernel]         [k]  futex_wait_setup
10.77%   [kernel]         [k]  _raw_spin_lock
 7.65%   [kernel]         [k]  futex_wait
 7.32%   libc -2.24.so    [.]  syscall
 6.67%   [kernel]         [k]  entry_SYSCALL_64_fastpath
 6.47%   [kernel]         [k]  get_futex_key_refs.isra.14
 4.03%   [kernel]         [k]  entry_SYSCALL_64
 3.79%   [kernel]         [k]  get_futex_key
 3.58%   [kernel]         [k]  do_futex
 3.48%   [kernel]         [k]  sys_futex
 2.31%   [kernel]         [k]  hash_futex
```

The benchmark v2, take three

```
       |                    a ^= (unsigned int) addr;
 5.67% |        xor    %edx,%eax
       |                m = ((u64)a * hm->pmul) >> 32;
12.57% |        mov    0x2e0(%rcx),%edx
 0.13% |        mov    %eax,%esi
17.12% |        imul   %rsi,%rdx
 3.63% |        shr    0x20,%rdx
       |                return (a - m * hm->prime) & hm->mask;
16.20% |        imul   0x2e4(%rcx),%edx
 3.37% |        sub    %edx,%eax
       |      hash_futex():
       |                return &mm->futex_hash.hash[slot];
 6.05% |        and    0x2e8(%rcx),%eax
 4.07% |        shl    0x6,%rax
 5.84% |        add    0x2f0(%rcx),%rax
```

v3

- **v3** `https://lkml.kernel.org/r/20160505204230.932454245@linutronix.de`
- A per process wide hash for all private futexes.
- The size of the hash can be pre-allocated. Otherwise one is allocated on first occasion.
- No auto-rehash. A sane default was used.
- Global hash as fallback if no hash can be allocated because. glibc does not tolerate errors here.

LINUTRONIX
LINUX FOR INDUSTRY

v3

- **v3** `https://lkml.kernel.org/r/20160505204230.932454245@linutronix.de`
- **A per process wide hash for all private futexes.**
- **The size of the hash can be pre-allocated. Otherwise one is allocated on first occasion.**
- **No auto-rehash. A sane default was used.**
- **Global hash as fallback if no hash can be allocated because. glibc does not tolerate errors here.**
- **Hash collision no good.**

Back to requirements

- Fit into existing model.
- Keep glibc interacting to a minimum.
- Guaranteed one hash bucket for each lock (collision free).
- ...

Further ideas

- FUTEX_ATTACH with ids / cookies.
- “attach” will return a cookie which is process wide valid.
- This cookie will be used instead of uaddr during futex operations.

Further ideas

- FUTEX_ATTACH with ids / cookies.
- “attach” will return a cookie which is process wide valid.
- This cookie will be used instead of uaddr during futex operations.
- pthread_mutex_init() could attach (but can’t fail).
- pthread_mutex_lock() could use the id then.
- The attached futexes need to be copied during fork(). urgh.

Further ideas, part two

- Every process adds two hash buckets to per-task pool.
- On each futex operation search of existing hb item for the address or take a new one from the pool.

Further ideas, part two

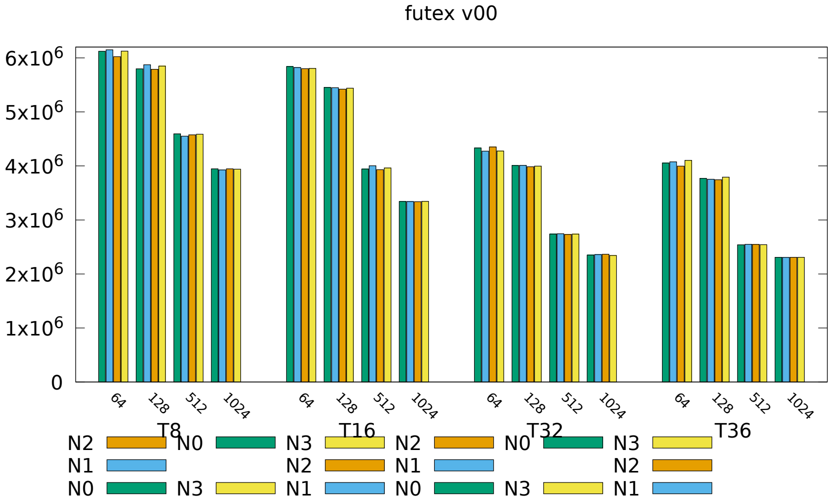
- Every process adds two hash buckets to per-task pool.
- On each futex operation search of existing hb item for the address or take a new one from the pool.
- Seems not to scale well.
- RBtree based lookup does not help, the global pool lock for hb and lookup is the problem.

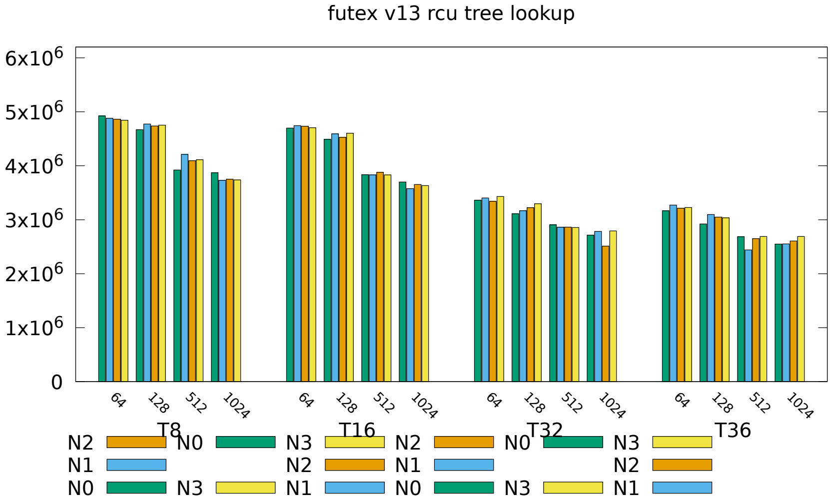
Further ideas, part three

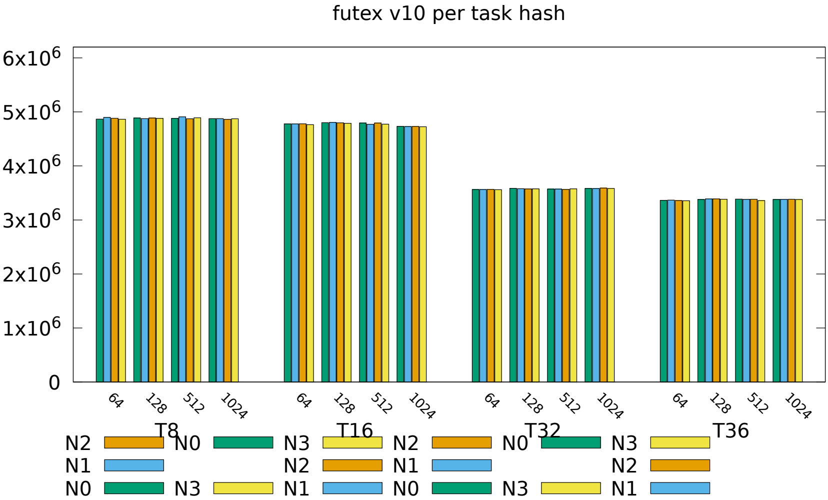
- FUTEX_ATTACH to attach a futex.
- Lookup uaddr $\rightarrow$ hb mapping via RBtree with RCU.

Further ideas, part three

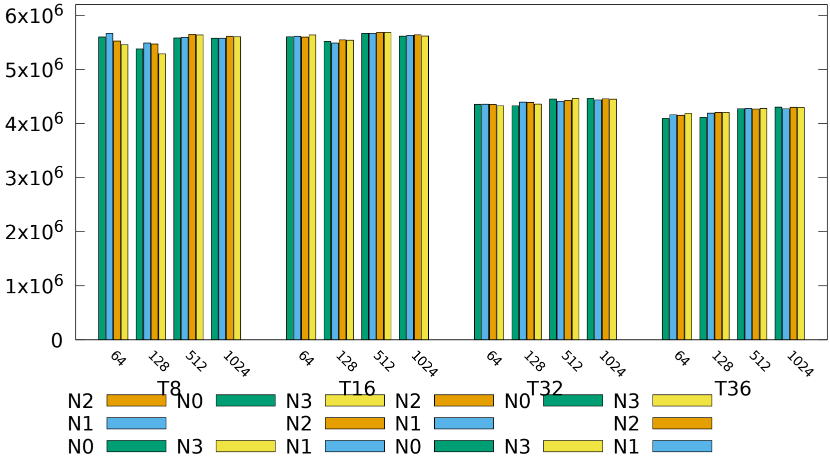
- FUTEX_ATTACH to attach a futex.
- Lookup uaddr → hb mapping via RBtree with RCU.
- Need attach support or attach on first use.
- Auto attach means no detach → unused memory.
- And this could be abused.


futex v00
6x10^6
5x10^6
4x10^6
3x10^6
2x10^6
1x10^6
0
64
128
512
1024
T8
T16
T32
T36
N2
N1
N0
N0
N3
N3
N2
N1
N2
N1
N0
N0
N3
N3
N2
N1


futex v13 rcu tree lookup
6x10^6
5x10^6
4x10^6
3x10^6
2x10^6
1x10^6
0
64
128
512
1024
T8
T16
T32
T36
N0
N1
N2
N3


futex v10 per task hash
6x10^6
5x10^6
4x10^6
3x10^6
2x10^6
1x10^6
0
64
128
512
1024
T8
T16
T32
T36
N0
N1
N2
N3

futex v12 unique ids



Thank you for your attention

Contact

Linutronix GmbH
Sebastian A. Siewior

Auf dem Berg 3
88690 Uhldingen
Germany

eMail bigeasy@linutronix.de