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

```c
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

- 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.
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

<table>
<thead>
<tr>
<th>Event Type</th>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>med sched_wakeup</td>
<td>comm=high</td>
</tr>
<tr>
<td>med sched_switch</td>
<td>prev=med/29 ==&gt; next=high/9</td>
</tr>
<tr>
<td>high sched_pi_setprio</td>
<td>comm=low oldprio=120 newprio=9</td>
</tr>
<tr>
<td>high sched_switch</td>
<td>prev=high/9 prev_state=S ==&gt; next=low/9</td>
</tr>
<tr>
<td>low sched_wakeup</td>
<td>comm=high prio=9</td>
</tr>
<tr>
<td>low sched_pi_setprio</td>
<td>comm=low oldprio=9 newprio=120</td>
</tr>
<tr>
<td>low sched_switch</td>
<td>prev=low/120 prev_state=R+ ==&gt; next=high/9</td>
</tr>
<tr>
<td>high sched_pi_setprio</td>
<td>comm=low oldprio=120 newprio=9</td>
</tr>
<tr>
<td>high sched_switch</td>
<td>prev=high/9 prev_state=D ==&gt; next=low/9</td>
</tr>
<tr>
<td>low sched_wakeup</td>
<td>comm=high prio=9</td>
</tr>
<tr>
<td>low sched_pi_setprio</td>
<td>comm=low oldprio=9 newprio=120</td>
</tr>
<tr>
<td>low sched_switch</td>
<td>prev=low/120 prev_state=R+ ==&gt; next=high/9</td>
</tr>
<tr>
<td>high sched_process_exit</td>
<td>comm=high prio=9</td>
</tr>
</tbody>
</table>
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 ==>
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+ ==>
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.
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”.
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 performance wise?

```
perf bench futex hash -f nfutex -n node -t nthreads
```

Performs an invalid FUTEX_WAKE over and over.
### The benchmark v2

<table>
<thead>
<tr>
<th>Percentage</th>
<th>Component 1</th>
<th>Component 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>23.13%</td>
<td>perf</td>
<td>[. ] workerfn</td>
</tr>
<tr>
<td>23.08%</td>
<td>[kernel]</td>
<td>[k] futex_wait_setup</td>
</tr>
<tr>
<td>21.46%</td>
<td>[kernel]</td>
<td>[k] entry_SYSCALL_64_fastpath</td>
</tr>
<tr>
<td>5.17%</td>
<td>[kernel]</td>
<td>[k] _raw_spin_lock</td>
</tr>
<tr>
<td>4.44%</td>
<td>[kernel]</td>
<td>[k] futex_wait</td>
</tr>
<tr>
<td>4.33%</td>
<td>libc – 2.24.so</td>
<td>[. ] syscall</td>
</tr>
</tbody>
</table>
The benchmark v2

```
for (i = 0; i < nfutexes; i++, w->ops++) {
    bb:    mov  nfutexes,%eax
    add    0x1,%ebx
    addq   0x1,0x18(%r12)
    cmp    %ebx,%eax
    ja     68
} while (!done);
```
The benchmark v2

The struct in question

```c
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

<table>
<thead>
<tr>
<th>Function</th>
<th>Percentage</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>futex_wait_setup</td>
<td>35.53%</td>
<td>[kernel]</td>
</tr>
<tr>
<td>_raw_spin_lock</td>
<td>11.54%</td>
<td>[kernel]</td>
</tr>
<tr>
<td>futex_wait</td>
<td>6.89%</td>
<td>[kernel]</td>
</tr>
<tr>
<td>libc — 2.24.so</td>
<td>6.70%</td>
<td>[] syscall</td>
</tr>
<tr>
<td>entry_SYSCALL_64_fastpath</td>
<td>6.11%</td>
<td>[kernel]</td>
</tr>
<tr>
<td>get_futex_key_refs.isra.14</td>
<td>6.09%</td>
<td>[kernel]</td>
</tr>
<tr>
<td>hash_futex</td>
<td>5.41%</td>
<td>[kernel]</td>
</tr>
<tr>
<td>entry_SYSCALL_64</td>
<td>3.79%</td>
<td>[kernel]</td>
</tr>
</tbody>
</table>
The benchmark v2, take two

```c
hash_futex()
{
    test 0x3,%al
    struct mm_struct *mm = current->mm;

    mov 0x2f8(%rdx),%rcx
    slot = key->private.address % mm->futex_hash.hash_bits;

    xor %edx,%edx
    mov (%rdi),%rax
    mov 0x2dc(%rcx),%esi
    div %rsi

    return &mm->futex_hash.hash[slot];

    shl 0x6,%rdx
    mov %rdx,%rax
    add 0x2e0(%rcx),%rax
}
```
### The benchmark v2, take three

<table>
<thead>
<tr>
<th>Function</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>futex_wait_setup</td>
<td>[k]</td>
<td></td>
</tr>
<tr>
<td>_raw_spin_lock</td>
<td>[k]</td>
<td></td>
</tr>
<tr>
<td>futex_wait</td>
<td>[k]</td>
<td></td>
</tr>
<tr>
<td>libc –2.24.so</td>
<td>[.]</td>
<td>syscall</td>
</tr>
<tr>
<td>entry_SYSCALL_64_fastpath</td>
<td>[k]</td>
<td>entry_SYSCALL_64_fastpath</td>
</tr>
<tr>
<td>get_futex_key.refs.isra.14</td>
<td>[k]</td>
<td>get_futex_keyrefs.isra.14</td>
</tr>
<tr>
<td>entry_SYSCALL_64</td>
<td>[k]</td>
<td>entry_SYSCALL_64</td>
</tr>
<tr>
<td>get_futex_key</td>
<td>[k]</td>
<td>get_futex_key</td>
</tr>
<tr>
<td>do_futex</td>
<td>[k]</td>
<td>do_futex</td>
</tr>
<tr>
<td>sys_futex</td>
<td>[k]</td>
<td>sys_futex</td>
</tr>
<tr>
<td>hash_futex</td>
<td>[k]</td>
<td>hash_futex</td>
</tr>
</tbody>
</table>
The benchmark v2, take three

\[
an ^= (\text{unsigned int}) \text{addr};
\]

```c
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   %rxi,%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
```
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.
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).
- ...

Sebastian A. Siewior  Linutronix GmbH
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 → 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 v13 rcu tree lookup
Thank you for your attention

Contact

Linutronix GmbH
Sebastian A. Siewior
Auf dem Berg 3
88690 Uhldingen
Germany

eMail bigeasy@linutronix.de