Futex Scaling in the Linux Kernel

January 2014
Davidlohr Bueso

Introduction to futexes (1/2)

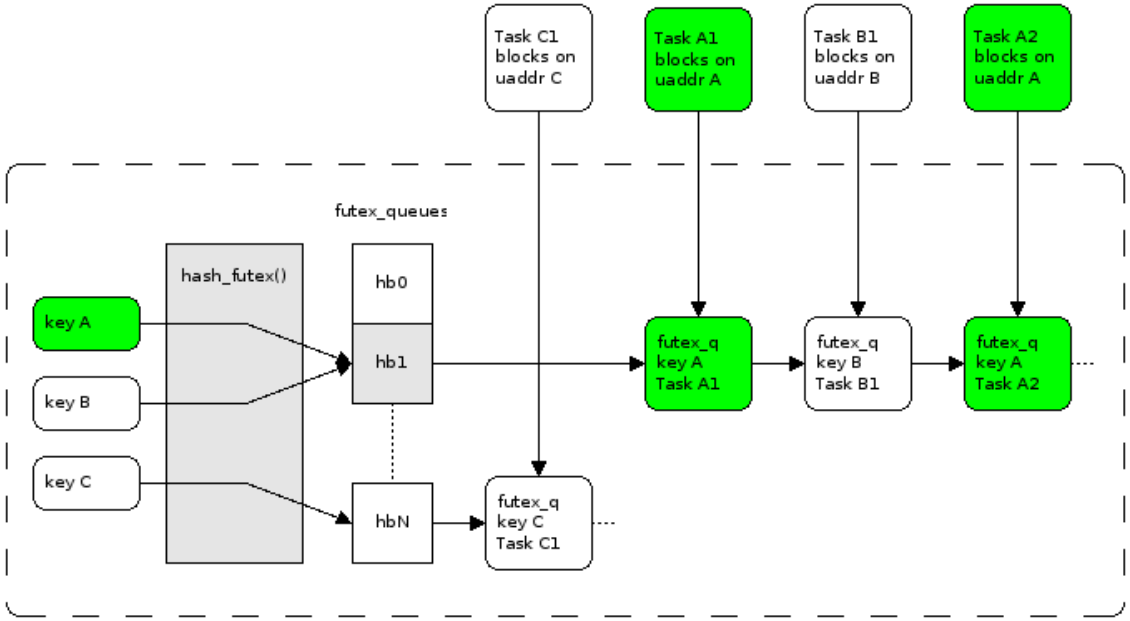
- Kernel functionality for userspace: “Fast user-space mutexes”
  - A futex is in essence a user-space address, e.g. a 32-bit lock variable field.
- man 2 futex:
  - “… method for a program to wait for a value at a given address to change, and a method to wake up anyone waiting on a particular address”
- Futexes are very basic and lend themselves well for building higher level locking abstractions such as POSIX threads:
  - pthread_mutex_*(), pthread_rwlock_*(), pthread_barrier_*()
  - pthread_cond_wait(), pthread_cond_signal/broadcast()
- But we already have SYSV IPC semaphores for user locking…. just look at Oracle.
  - The F in Futex stands for fast: in the uncontended cases, the kernel is never aware and no need to mode switch from userland.
  - In the case of IPC this is not true as jumping to kernel space is always required to handle the call.

Introduction to futexes (2/2)

- Three kinds of futexes:
  - Regular, Priority Inheritance (PI) & Robust
  - PI and robust futexes are exceptions to the user-defined-policy rule regarding the state variable. Their state depends not only on the locked state of the mutex, but also on the identity of the owner and whether or not there are waiters.
    - For instance, if a program crashes while holding a lock then waiters need to be notified that the lock owner exited in some irregular way.
- Introduced by Franke, Kirkwood & Russell (IBM) in 2002.
  - *Fuss, Futexes and Furwocks: Fast Userlevel Locking in Linux*

General Architecture (1/2)

- Wait queues (priority linked lists) are at the heart of futexes.
  - Governed by a chained global hash table.
  - The uaddr is used by the kernel to create a unique futex key, each key hashes to a hash bucket.
  - Each bucket is serialized by a spinlock – all operations require holding the lock beforehand.
  - One or more futexes can share the queue (collisions).



General Architecture (2/2)

- This architecture is common for all kinds of futexes.
- Depending on the userspace program's needs, futexes can be shared or private. This is particularly important for performance when computing the hash key:
  - Private (pthread_create)
    - Particularly fast when computing the key.
    - uaddresses are distinguished by their virtual address (same *->mm* addr space).
    - *hb =hashfn(*uaddr, current->mm*)*
  - Shared (fork)
    - Traditional approach to compute the key.
    - Needs to take *mmap_sem*: cacheline pollution, lock contention, depends heavily on mm/
    - *hb = hashfn(page->index, file_inode(vma->vm_file))*

Bottlenecks

- All issues impact all types of futexes.
- Futexes currently suffer from its original design: a unique, shared hash table.
  - For NUMA systems, all memory for the table is allocated on a single node.
  - What's worse is that the size is ridiculously small (256 hash buckets).
  - Both problems hurt scalability, considerably.

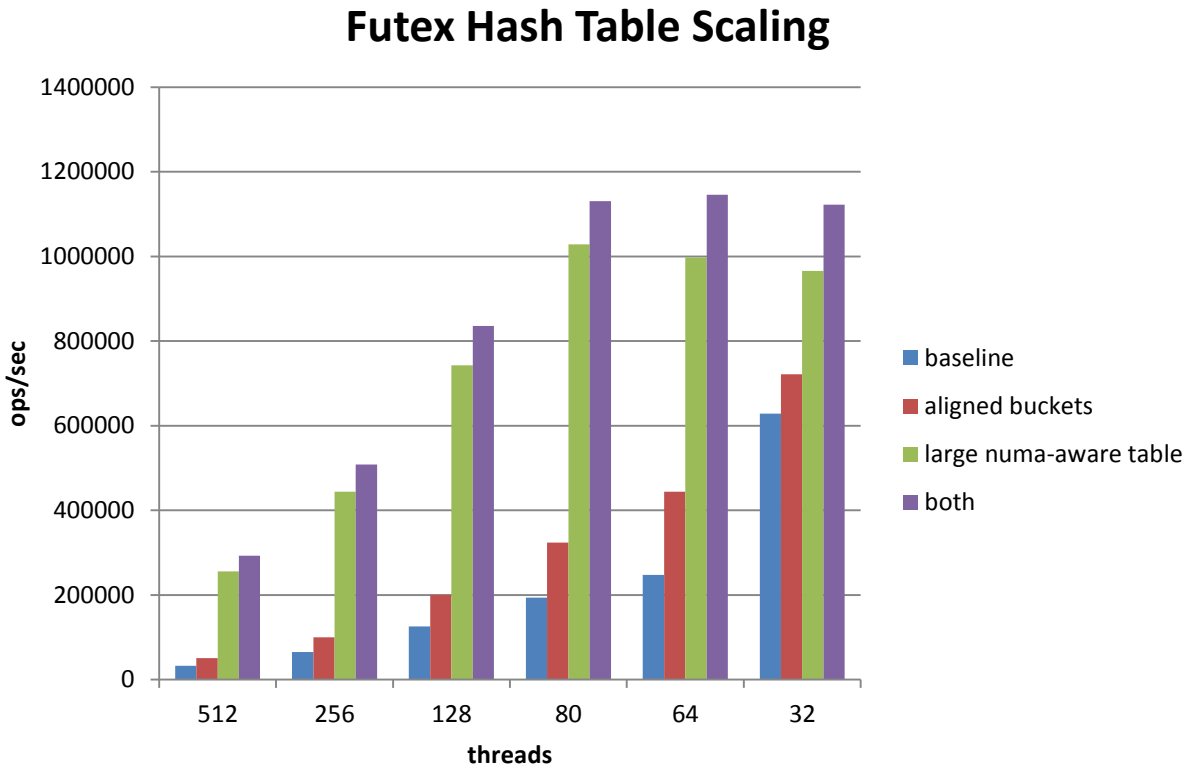
- Critical Regions can become quite large. The following operations are done while holding the *hb->lock*:
  - Task, mm/inode refcounting.
  - Wake up tasks.
  - Plist handling.

Scaling Patches

Larger, NUMA-aware Hash Table (1/2)

- 256 hash buckets per CPU.
  - 256 * nCPUs cacheline aligned hash buckets
  - Less collisions and more spinlocks leading to more parallel futex call handling.
- Distribute the table among NUMA nodes instead of a single one.
- As expected, the benefits of this patch become more evident as more futexes are used. On a 1Tb, 80-core, 8-socket Proliant DL980:
  - 1024 * 32 futexes -> ~78% throughput increase.
  - 1024 * 512 futexes -> ~800% throughput increase.
- The perfect hash size will of course have one to one hb:futex ratio.
- Distribution of bucket sizes. Normal distribution indicates that the hash function spreads objects evenly among the hash buckets.

NUMA-aware Larger Hash Table (2/2)



Lockless Waitqueue Size (1/2)

- In FUTEX_WAKE, there's no reason to take the *hb->lock* if we already know the list is empty and thus one to wake up. Depending on the workload and system, this can drastically reduce contention on the spinlock.
- There is a huge caveat to this:
  - A racy window exists between the futex_wait call and when the task is actually added to the priority list (plist).
  - If all possible waiters aren't acknowledged by the time we do the checks, tasks can sleep/block forever.
  - Ordering guarantees must be preserved. Ensure that waiters either observes the changed user space value before blocking or is woken by a concurrent waker.
    - Overhead for this is minimal – measured latency of nthread wakeups, 1 at a time.

Lockless Waitqueue Size (2/2)

- Two solutions to this:
  1. Use a separate atomic counter to keep track of the list (original approach).
     - Exact results.
     - Very invasive, kind of redundant, atomic counters!! (we do already dirty the cacheline anyways so it's not too bad), enlarges the structure on 32-bit kernels.
  2. Couple plist head empty with is hb->lock taken(?) checks.
     - Tasks trying to enter the critical region are most likely potential waiters that will be added to the plist.
     - Cheaply solves the race and doesn't add any extra infrastructure to futexes.
     - Can produce potential false positives when the lock is taken by a waker path.

Others

- Looking at reducing the FUTEX_WAKE critical regions by allowing tasks to wake after they've released the *hb->lock*.
  - Can also benefit SYSV semaphores.
  - Very tricky as it can cause spurious wakeups and we have to make very sure that the entire kernel can handle this behavior – it currently does not.
- Can could replace tsk refcounting (get_task_struct(p)/put_task_struct(p)) with RCU – in the wake up path, the task won't go away if it's RCU-delayed.
- Jason looked at adding a futex content vs passed value check in FUTEX_WAIT paths before taking the lock.

Backup Slides

perf-bench futex (1/2)

- Darren Hart's futextest suite does not provide enough performance information at a finer granularity.
  - This is best for developers that are more interested in using futexes in their applications, and not hacking at them in the kernel. Ie: mutex/rwlock implementations.
  - It does provide a good amount of functional/unit testing. Could use more, though.
  - Outside of kernel tree, less attention – last commit was in 2011.

- Futex microbenchmarks are ideal for perf-bench (Available on lkml).
  - perf bench futex [<operation> <all>]
- Measures latency of different operations:
  - Futex hash
  - Futex wake
  - Futex requeue/wait

perf-bench futex (2/2)

$ **perf bench futex wake**
# Running 'futex/wake' benchmark:
Run summary [PID 4028]: blocking on 4 threads (at futex 0x7e20f4),
waking up 1 at a time.

[Run 1]: Wokeup 4 of 4 threads in 0.0280 ms
[Run 2]: Wokeup 4 of 4 threads in 0.0880 ms
[Run 3]: Wokeup 4 of 4 threads in 0.0920 ms
[Run 4]: Wokeup 4 of 4 threads in 0.0920 ms
[Run 5]: Wokeup 4 of 4 threads in 0.0870 ms
[Run 6]: Wokeup 4 of 4 threads in 0.0820 ms
[Run 7]: Wokeup 4 of 4 threads in 0.0210 ms
[Run 8]: Wokeup 4 of 4 threads in 0.0880 ms
[Run 9]: Wokeup 4 of 4 threads in 0.0990 ms
[Run 10]: Wokeup 4 of 4 threads in 0.0260 ms
Wokeup 4 of 4 threads in 0.0703 ms (+-14.22%)

$ **perf bench futex hash**
# Running 'futex/hash' benchmark:
Run summary [PID 4069]: 4 threads, each operating on 1024 futexes for 10 secs.

[thread 0] futexes: 0x1982700 ... 0x19836fc [3507916 ops/sec]
[thread 1] futexes: 0x1983920 ... 0x198491c [3651174 ops/sec]
[thread 2] futexes: 0x1984ab0 ... 0x1985aac [3557171 ops/sec]
[thread 3] futexes: 0x1985c40 ... 0x1986c3c [3597926 ops/sec]

Averaged 3578546 operations/sec (+- 0.85%), total secs = 10

Funny (yet true) Quotes

- "futexes are tricky" – Ulrich Drepper
- "The futexes are also cursed." – Header comment in *kernel/futex.c*
- "Is it worth the introduction of atomic operations into a file known to the state of California to increase the risk of liver failure? " – Darren Hart [https://lkml.org/lkml/2013/11/23/3]

References

- Drepper, Ulrich. "Futexes are Tricky". Nov 2011.
- Hart, Darren. "A futex overview and update". lwn.net. Nov 2009.
- Hart, D., Guniguntala, D. "Requeue-PI: Making Glibc Condvars PI-Aware". Proc. RT Linux Summit 2011.
- Bueso, Davidlohr. "futex: Wakeup optimizations". lwn.net/lkml. Dec 2013 and Jan 2014.