Reentrant kmalloc for any context

Alexei Starovoitov

Existing slab wrappers

- bpf_mem_alloc
- kretprobe objpool
- ...

BPF's preallocated past...

- 11 years ago the only BPF use cases were networking and tracing
  - tracing context is unknown. NO kmalloc
  - networking context is typically BH. Ok to kmalloc
- By default BPF maps are preallocated
  - all elements of a hash table are preallocated in per-cpu freelists
  - Ex: max_entries=10k hashtab on 100 cpus will have 100 elements in each per-cpu freelist
  - push/pop can happen on different cpus, simple round robin stealing from other cpu-s
  - freed elements are instantly reused
  - high performance
- Hash map rarely used at full capacity to maintain O(1) performance
  - Lots of preallocated memory is unused

BPF's preallocated past...

- networking could use BPF_F_NO_PREALLOC to mitigate memory waste
    - kmalloc(GFP_ATOMIC), call_rcu() in free path
        - can have memory spikes
    - performance could be several times slower depending on usage
    - rarely used in practice
    - cannot be used when tracing and networking share a map

bpf_mem_alloc to the rescue

- kmalloc-like per-cpu buckets {96, 192, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096}
- NMI-safe per-cpu freelists with low/hi watermark
    - average of 64 per-cpu elements for smaller buckets
- Once below low watermark raise irq_work to refill
- High performance
- Global bpf_mem_alloc with buckets
- kmem_cache_create()-like bpf_mem_alloc for fixed size elements

bpf_mem_alloc issues

- allocations from irq disabled context are unpredictable
- irq_work_queue() is/was broken on some arm64
- Memory is pinned in free list for bpf system only
    - each bpf hash map uses its own one size bpf_mem_alloc instance
    - Rest of bpf uses global bpf_mem_alloc
- Not as bad memory waste as full prealloc, but
    - global bpf_mem_alloc mirrors kmalloc-* slubs
    - fixed size bpf_mem_allocs mirror kmem_cache_create-d slubs
    - merging is needed
- Do not want to reinvent slub features one by one

bpf_mem_alloc issues

- async refill is ok for small sizes
    - 1k+ objects have infeasible trade-off :
        - either excessive memory waste
        - or sporadic allocation failures
- kmalloc is synchronous. Good.

Synchronous kmalloc stack

- kmalloc
  - slab_alloc_node
    - cmpxchg16
      - __slab_alloc
        - new_slab
          - alloc_pages
            - rmqueue_pcplist
              - __rmqueue
                - wakeup_kswapd // when __GFP_KSWAPD_RECLAIM is set

Synchronous kmalloc stack

- kmalloc
  - slab_alloc_node
    - cmpxchg16
      - __slab_alloc
        - new_slab
          - alloc_pages
            - rmqueue_pcplist
              - __rmqueue
                - wakeup_kswapd // when __GFP_KSWAPD_RECLAIM is set

try_alloc_pages cover these steps

try_alloc_pages implementation details

- rmqueue_pcplist
  - already spin_trylock(). No changes were necessary.
- __rmqueue
  - convert spin_lock_irqsave to spin_trylock_irqsave if (alloc_flags & ALLOC_TRYLOCK)
- try_charge_memcg
  - convert local_lock_irqsave to localtry_trylock_irqsave if (gfpflags_allow_spinning(gfp_mask))

```
static inline bool gfpflags_allow_spinning(const gfp_t gfp_flags)
{
        /*
         * !__GFP_DIRECT_RECLAIM -> direct claim is not allowed.
         * !__GFP_KSWAPD_RECLAIM -> it's not safe to wake up kswapd.
         * All GFP_* flags including GFP_NOWAIT use one or both flags.
         * try_alloc_pages() is the only API that doesn't specify either flag.
         *
         * This is stronger than GFP_NOWAIT or GFP_ATOMIC because
         * those are guaranteed to never block on a sleeping lock.
         * Here we are enforcing that the allocation doesn't ever spin
         * on any locks (i.e. only trylocks). There is no high level
         * GFP_$FOO flag for this use in try_alloc_pages() as the
         * regular page allocator doesn't fully support this
         * allocation mode.
         */
        return !!(gfp_flags & __GFP_RECLAIM);
}
```

Reentrant kmalloc for any context

- kmalloc
  - slab_alloc_node
    - cmpxchg16
      - __slab_alloc
        - new_slab
          - alloc_pages
            - rmqueue_pcplist
              - __rmqueue

try_kmalloc

try_alloc_pages cover these steps

Plan so far

- in get_freelist()... support __CMPXCHG_DOUBLE only (for now)
    - punt on try-locking bit_spin_lock to later
    - essentially no changes in fast path
- s/local_lock/localtry_trylock/ in slow path
- if pfmemalloc mismatch return ENOMEM
- in ___slab_alloc() return ENOMEM when trylock fails
    - ignore node mismatch ? pretend to be NUMA_NO_NODE
    - don't use get_partial()
- new_slab() -> try_alloc_pages()
- kmem_cache_debug() will work as-is with trylock