KVM Dirty Page Tracking

Peter Xu <peterx@redhat.com>, Red Hat

Outlines

- Concepts
- Migration & Challenges
- Bitmap copy & atomics

Concepts

What is Dirty Tracking

- Goal: tracking guest memory changes for different reasons
- Types of dirty tracking
  - Synchronous: tracee is blocked
    - Examples: shadow pgtable tracking, VM live snapshots
  - Asynchronous: tracee is not blocked
    - Example: migration, dirty rate measurements

Synchronous Dirty Tracking

- KVM guest page table tracking
  - Used by shadow paging only, two-dimensional paging not needed
  - Invalidate shadow pgtable (L1+L2) when guest pgtable (L1) changes
  - KVM internal interface, usable for kernel drivers only (kvmgt)
- VM live snapshots
  - Snapshot guest device states at a single point in time
  - Based on uffd-wp, only anonymous memory supported
  - Similar to migration, but track dirty page synchronously

Asynchronous Dirty Tracking

- It's all about migration

- Step 1: Trapping
  - Write-protection
  - PML (Page Modification Logging)

- Step 2: Reporting
  - Dirty log, per-vm, bitmap-based
  - Dirty ring, per-vcpu ring-based

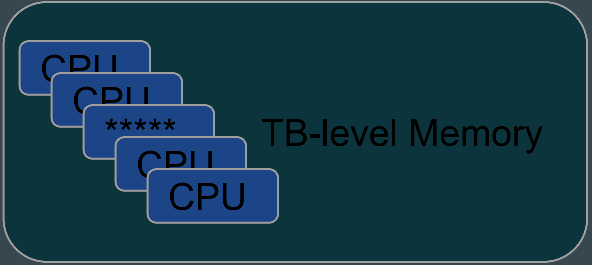
Migrations & Challenges

VM Migrations

- Upstream KVM is evolving with more efficient migrations
  - Keqian’s work on lazy wr-protect of huge pages
    - Further speedup KVM_SET_USER_MEMORY_REGION with init-all-set
  - Ben’s work on the new tdp mmu
    - MMU lock => rwlock, concurrent page faults (including dirty tracking)
  - KVM dirty ring landed
    - Linux v5.11 (Feb 2021) / Qemu v6.1 (July 2021, initial support only)
- QEMU needs to catch up!
- What’s next? “Huge VM migration”

Huge VM Migrations

- Properties
    - More vCPUs (100+)
    - More memories (TB-level+)
    - Have serious & important workloads
    - Quality assurance, even during migration
- Challenges
    - Existing algorithms/structures not scaling
    - Auto-converge not applicable any more
    - Hugetlbfs



Huge VM Migrations - Challenges & Solutions

- Issue 1: Not-scaling algorithms/structures
  - Long term effort in both QEMU & KVM, already getting better!
- Issue 2: Convergence
  - Postcopy required
- Issue 3: Data copy bottleneck
  - Multi-FD, or
  - setsockopt(MSG_ZEROCOPY), or
  - Both!

Huge VM Migrations - Challenges & Solutions

- Issue 4: High downtime during postcopy (hugetlbfs)
  - Double-map of hugetlbfs can reduce page fault latency
  - Allows hugetlbfs 2M/1G pages to be mapped in smaller chunk, e.g. 4K
  - Merge small pages into huge pages when finished
  - Still not available yet
- Issue 5: High downtime switching from precopy to postcopy
  - Userfaultfd minor-mode (contributed by Google, merged v5.13 in 2021)
  - Allows dest VM runs earlier on stalled pages
  - No anonymous support, but support shmem/hugetlbfs
  - QEMU may need a new madvise() to zap pgtable but keep page caches

Example: Bitmap Copy

- What we measured (from QEMU)
    - Sync dirty bitmap took ~200ms for not-so-busy 3TB guest (~100MB bitmap)
- Reasons
    - Three layers of bitmap: kvm slot, ram_list.dirty_memory, migration
    - Different devices have standalone bitmaps: kvm, vhost, vfio, ...
    - Copy bitmaps using xchg()/atomics for thread safety
- Need to look into
    - Merging/reducing bitmap layers/operations
    - Copy bitmaps more efficiently (next slide)

Bitmap Copy & Atomics

- Atomic ops are heavily used in dirty bitmap operations for thread-safety
- Atomic ops are not so cheap
  - Memory-bus lock required
- Compare xchg() v.s. normal memory copy (measured on i7-8665U)
  - All cache-hit in L1 (e.g. xchg() on single value): **8x** slower
  - All cache-miss in L3 (e.g. xchg() a bitmap larger than L3 cache): **3x** slower (More data in next slide)

Bitmap Copy w/ xchg()

- Copy bitmap for 8TB memory (256MB bitmap):

CPU Model	xchg()	Memory copy	Ratio
Intel(R) Core(TM) i7-8665U CPU @ 1.90GHz	240ms	80ms	3x slower
Intel(R) Xeon(R) CPU E5-2630 v4 @ 2.20GHz	525ms	148ms	3.5x slower

[Test case: https://github.com/xzpeter/clibs/blob/master/bsd/bitmap.c]

Bitmap Copy - KVM Side

- KVM does not have such issue (at least not a major one)
  - With CLEAR_LOG, we do copy_to_user(bitmap) without xchg()
  - When re-protect, xchg() used, overhead buried in pgtable walks (?)

```
@@ -2968,8 +2979,21 @@ void mark_page_dirty_in_slot()
            if (kvm->dirty_ring_size)

kvm_dirty_ring_push(kvm_dirty_ring_get(kvm),
                                        slot, rel_gfn);
-               else
+               else {
+                       lockdep_assert_held(&kvm->mmu_lock);
                        set_bit_le(rel_gfn, memslot->dirty_bitmap);
+               }
        }
 }
 EXPORT_SYMBOL_GPL(mark_page_dirty_in_slot);
```

(a) Set dirty

```
@@ -1804,14 +1804,19 @@ static int
kvm_get_dirty_log_protect()
                for (i = 0; i < n / sizeof(long); i++) {
-                       unsigned long mask;
+                       unsigned long mask = dirty_bitmap[i];
                        gfn_t offset;

-                       if (!dirty_bitmap[i])
+                       if (!mask)
                                continue;

                        flush = true;
-                       mask = xchg(&dirty_bitmap[i], 0);
+                       dirty_bitmap[i] = 0;
                        dirty_bitmap_buffer[i] = mask;

                        offset = i * BITS_PER_LONG;
```

(b) Get dirty (without CLEAR)

```
@@ -1917,12 +1922,18 @@ static int kvm_clear_dirty_log_protect()
-               unsigned long mask = *dirty_bitmap_buffer++;
-               atomic_long_t *p = (atomic_long_t *) &dirty_bitmap[i];
+               unsigned long mask = *dirty_bitmap_buffer++, tmp;
                if (!mask)
                        continue;

-               mask &= atomic_long_fetch_andnot(mask, p);
+               tmp = dirty_bitmap[i];
+               dirty_bitmap[i] &= ~mask;
+               mask &= tmp;
```

(c) Clear dirty (with CLEAR)

Bitmap Copy - Summary

- QEMU may need a rework on copying/merging bitmaps
- Solution: rwlock + atomics?
  - When set dirty:
    - read_lock() + atomics (atomic ops avoid concurrent read races)
  - When collect/copy dirty
    - write_lock() + memcpy(): write lock avoids all races
- Rwlock read_lock()/write_lock() contain memory barriers by nature
- Need to verify and test

Comments welcomed, thanks!