Fast Write Protection

Xiao Guangrong
<xiaoguangrong@tencent.com>

Agenda

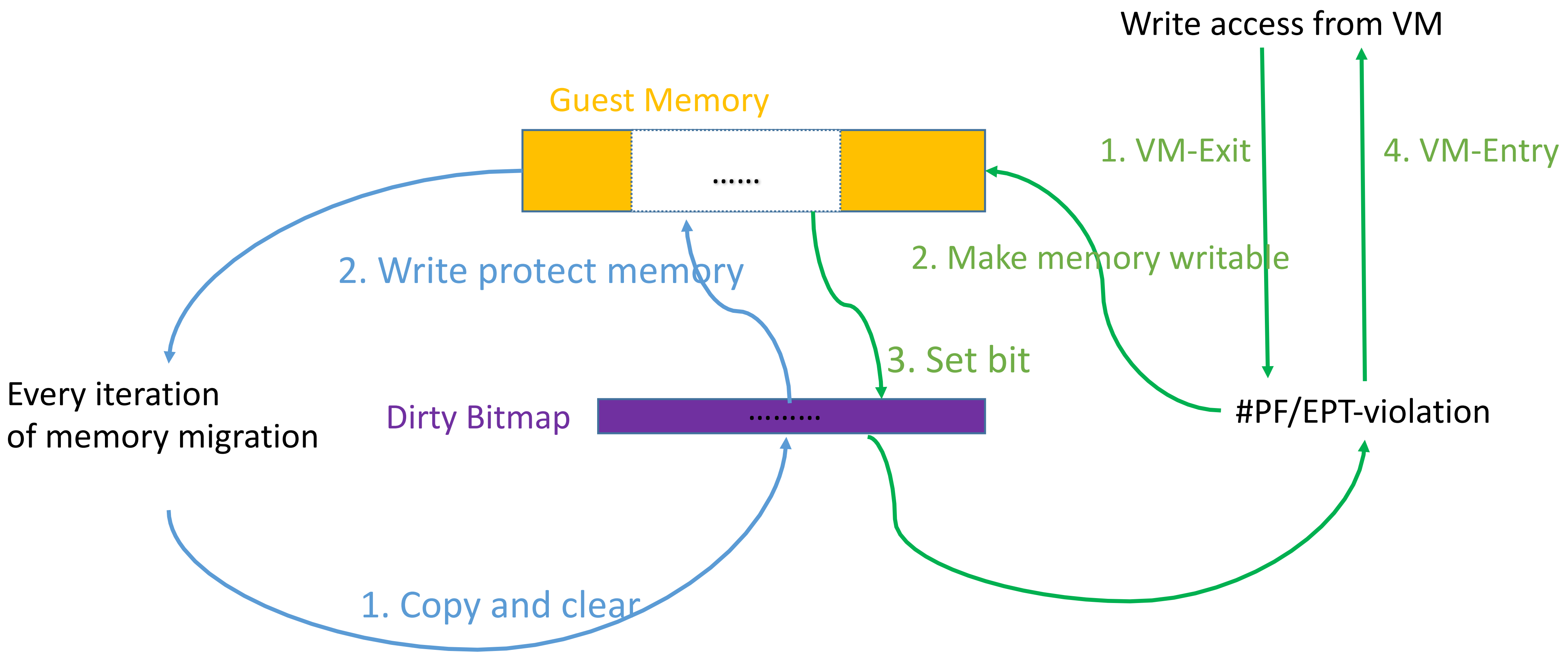
- Background
- Challenges
- Fast write protection
- Dirty bitmap
- Evaluation
- Future plan

Background

- Live migration is a key feature for cloud provider, e.g., Tencent Cloud
    - Load Balance
    - Error recovery
    - Maintainability
    - Etc.

Background (Cont.)

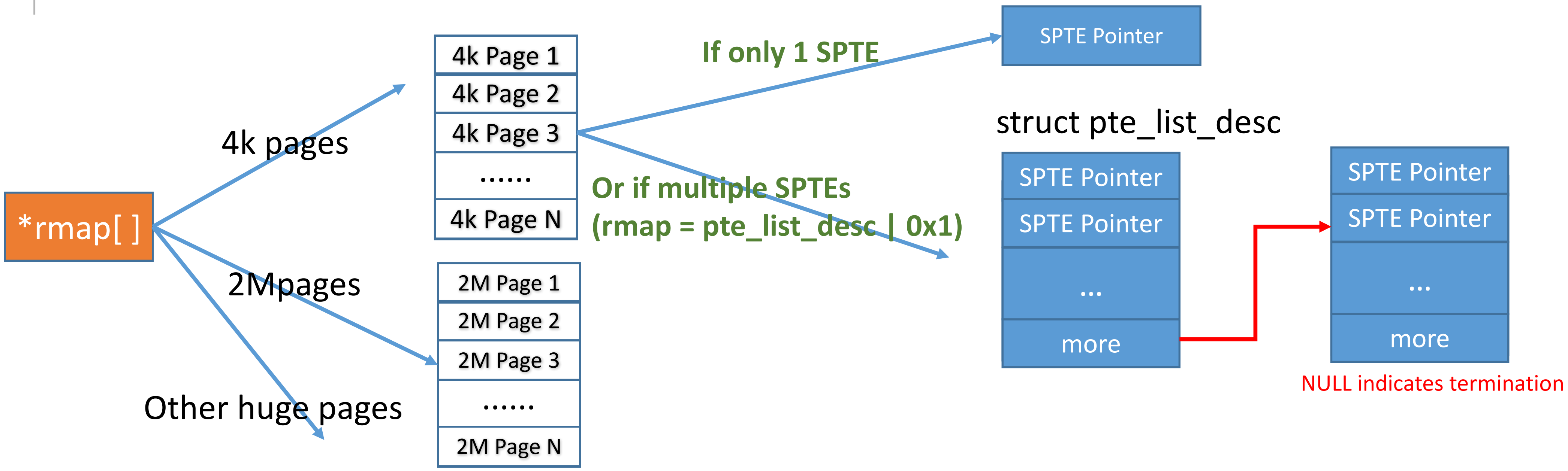
- Write protection is a key performance dependence for Live migration



Challenges

- Current write protection implantation
  - It is based on SPTE RMAP (Shadow Page Table Entry Reverse MAPping)

```
struct kvm_arch_memory_slot {
        struct kvm_rmap_head *rmap[KVM_NR_PAGE_SIZES];
        struct kvm_lpage_info *lpage_info[KVM_NR_PAGE_SIZES - 1];
        unsigned short *gfn_track[KVM_PAGE_TRACK_MAX];
};
```

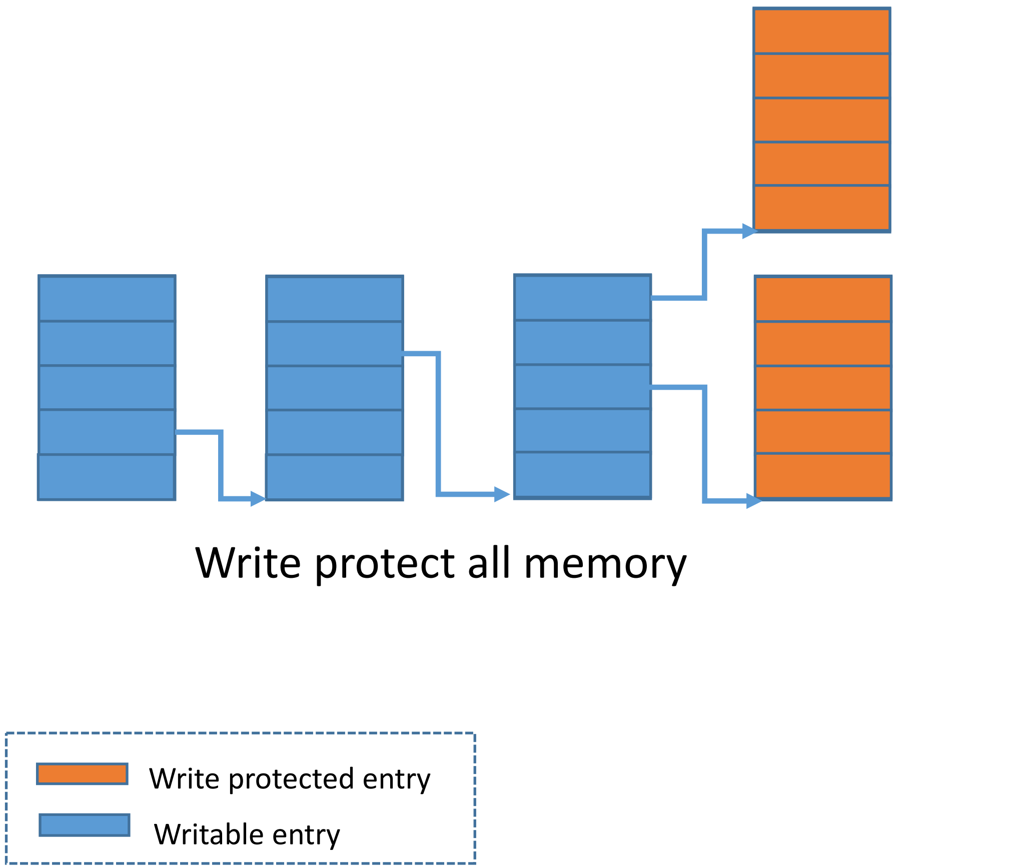


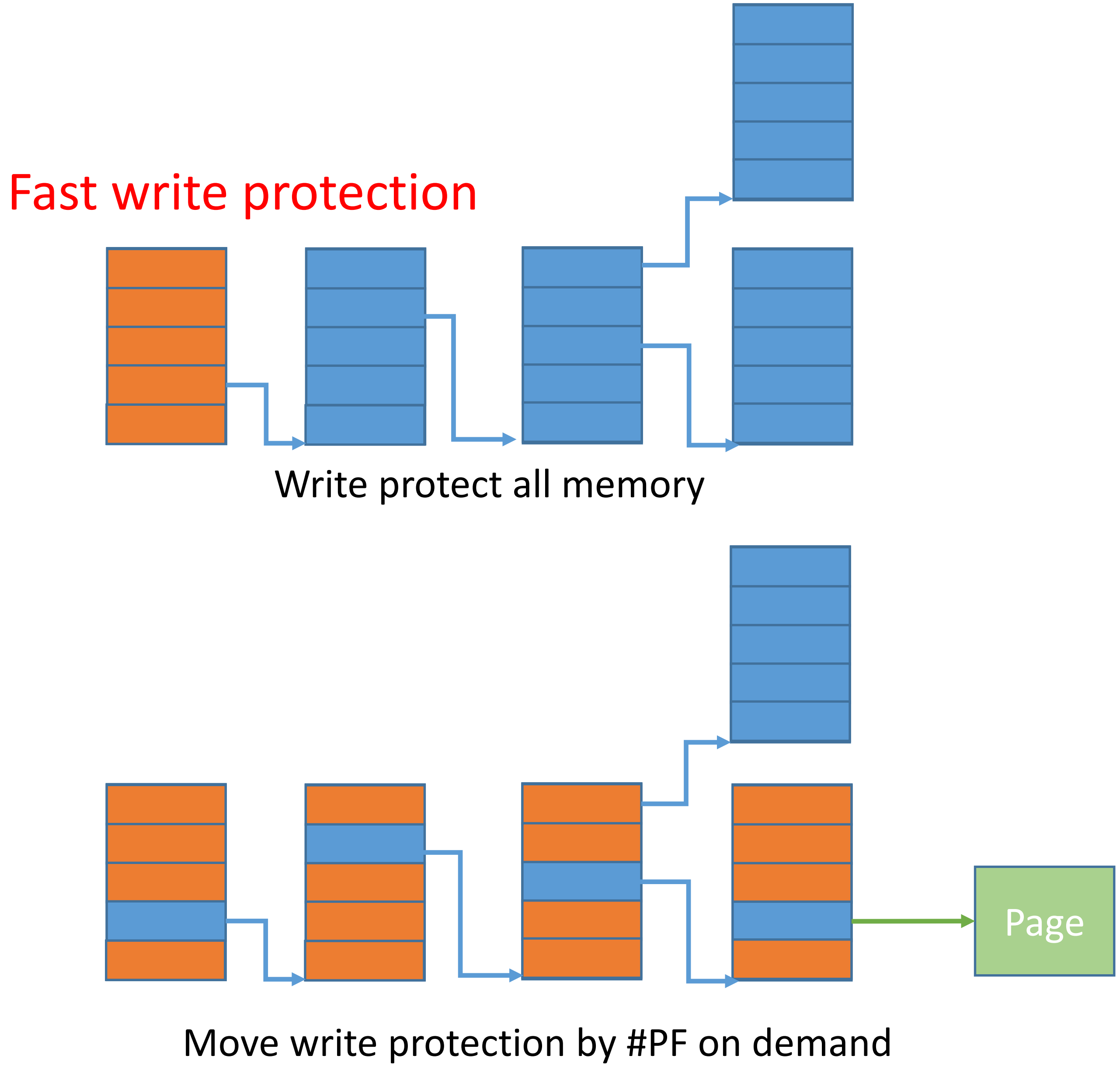
Challenges (Cont.)

- It traverses rmaps of all memslots and makes spte readonly one by one
  - It is not scalable as it depends on the size of memory in VM
- More worse, it needs to hold mmu-lock
  - Mmu-lock is a big & hot lock as It is contended by all vCPUs to update shadow page table

Fast write protection

- Overview





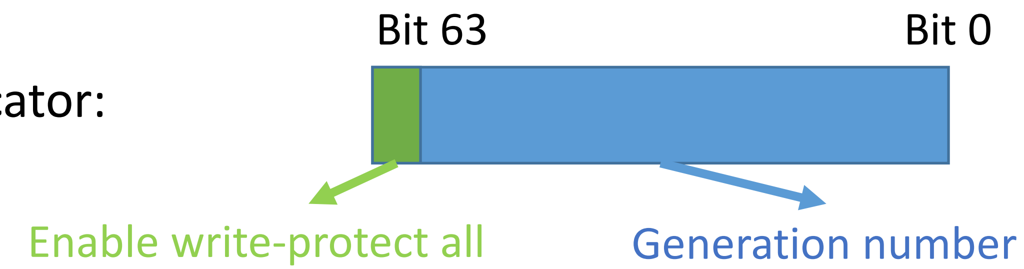
Fast write protection (Cont.)

- The basic idea was raised by Avi Kivity in ~2011 during my vMMU development
- Extremely fast
- The O(1) algorithm
    - Not depend on the capacity of guest memory
- Lockless
    - Not require mmu-lock
    - Not hurt the parallel of vCPUs

Fast write protection: Implementation

- A new API, KVM_WRITE_PROTECT_ALL_MEM, is introduced
- A global write-protect indicator is introduced
  - In order to make it lockless, the indicator is split to two parts

Global write-protect indicator:



- A write-protect-all generation number is introduced to shadow page table (struct kvm_mmu_page)
  - Which is synced with global generation number and used to check if write protection is needed

Fast write protection: Implementation (Cont.)

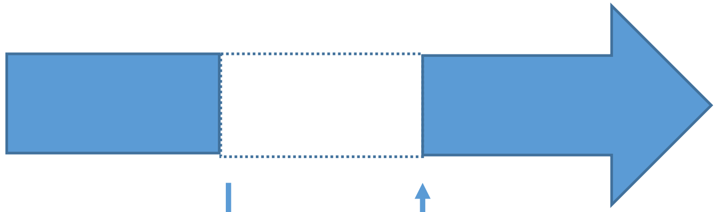
Migration Thread

Ioctl(KVM_WRITE_PROTECT_ALL_MEM)

Global-gen-num++

Kick off all vCPUs and ask them to
Reload its root page table

vCPU



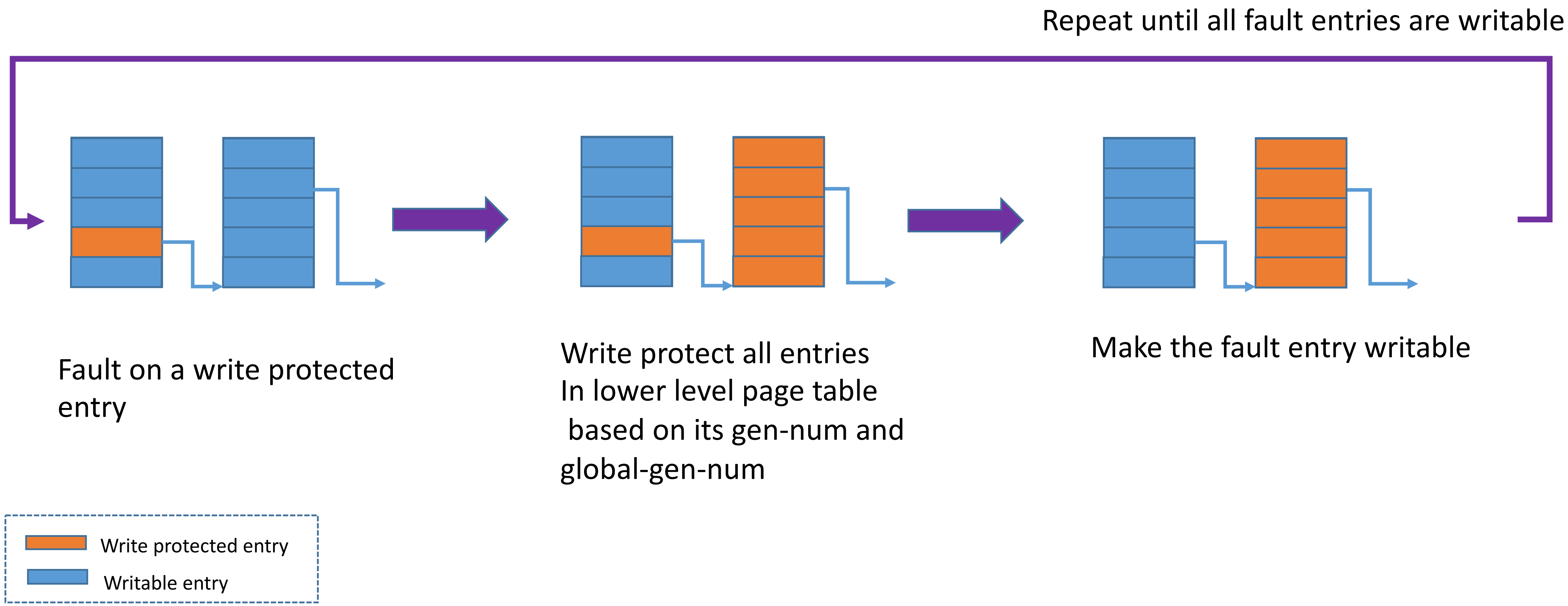
VM-Exit

VM-Entry

Reload root page table:
    if (gen-number of shadow page != global–gen-num) {
        write protect all entries
        update shadow page’s gen-num
    }

Fast write protection: Implementation (Cont.)

- For page fault handler



Fast write protection: Implementation (Cont.)

- For the new created shadow page, we can simply set its write-protect generation number to global generation
- To speed up the process which makes all entries of the shadow page readonly, we introduce these new stuffs to shadow page table
  - possible_writable_spte_bitmap which indicates the writable sptes
  - possiable_writable_sptes which is a counter indicating the number of writable sptes in the shadow page

Dirty bitmap

- One call of KVM_WRITE_PROTECT_ALL_MEM can write protect all VM memory, so that KVM_GET_DIRTY_LOG need not do write protection anymore
- A new flag is introduced to KVM_GET_DIRTY_LOG to ask KVM skipping write protection
  - KVM_DIRTY_LOG_WITHOUT_WRITE_PROTECT
- In fact, that opens the opportunities to speed up KVM_GET_DIRTY_LOG
  - Now, it just copies the bitmap from kernel to userspace

Dirty bitmap: omit KVM_GET_DIRTY_LOG

- Make the bitmap be shared between userspace and KVM
- Userspace & KVM async-ly and atomic-ly operate the bitmap, i.e., move the operation in current KVM_GET_DIRTY_LOG to userspace

Userspace

```
Fetch bitmap:
for (i = 0; i < n / sizeof(long); i++) {
   mask = xchg(&dirty_bitmap[i], 0);
  Saved_dirty_bitmap_buffer[i] = mask;
}
```

KVM

```
mark_page_dirty:
     set_bit_le(gfn_index, memslot->dirty_bitmap);
```

- Avoiding xchg is also possible (by introducing double dirty bitmaps and switch them during fetching dirty bits?)

Evaluation

- When we did the evaluation, shared bitmap has not been implemented yet
- The following cases are based on the VM which has 3G memory + 12 vCPUs
- Case 1: evaluate the time for KVM_GET_DIRTY_LOG

	Before	**After**	**Result**
Time (ns)	64289121	137654	+46603%

Evaluation

- Case 2: evaluate the time to make all memory writable after write-protection

	Before	After	Result
Time (ns)	281735017	291150923	-3%

- Performance drop due to
  - a) **fast page fault** which locklessly fix #PF on last level of shadow page, so before our work, it is complete lockless, after our work, need mmu-lock to make upper levels writable
  - b) need little time to move write protection from upper levels to lower levels
- We think it is acceptable, particularly, mmu-lock contention (caused by write protection) did not take into account for this case

Evaluation (Cont.)

- The following cases are for the VM which has 30G memory and 8 vCPUs, during live migration, a memory benchmark is running in the VM which repeatedly writes 3000M memory
- Case 3: for the new booted VM, that means, mmu-lock is required to map physical memory into shadow page table

	Before	After	Result
Dirty page rate (pages)	333092	497266	+49%
Total time of live migration	12532	18467	-47%

- As fast write protection reduces the contention of mmu-lock, VM writes memory more efficiently than before
- No surprise, as more dirty pages are generated, more time is needed to migrate memory

Evaluation (Cont.)

- Case 4: for the pre-written VM, that means, all memories are mapped in, fast page fault can directly make the page table writeable without holding mmu-lock on the last level

	Before	After	Result
Dirty page rate (pages)	447435	449284	+0%
Total time of live migration	31068	28310	+47%

- We also noticed that the time of dirty log for the first time, before our work is 156 ms, after our work, only 6 ms is needed

Future plan

- Currently, v2 of fast write protection has been posted out
  - https://lkml.org/lkml/2017/6/20/274
- Ask Paolo, Marcelo, Radim and other guys to comment on it and push it to upstream
- Enable it on QEMU side
- Think shared dirty bitmap carefully and enable it
- Others…

Q/A?

Thanks!