Giant VM DSM 프로토콜 기초 자료

2021.08.09

Background

- *GiantVM DSM이 사용하는* **ivy protocol 기초**

(인용 자료: Distributed Resource Management: Distributed Shared Memory CS-550: Distributed Shared Memory [Si. S ' 94], https://slidetodoc.com/distributed-resource-management-distributed-shared-memory-cs550-distributed)

Coherence protocol

- Page access modes: read only, write, nil (invalidate)
- Multiple readers-single writer semantics

- Protocol overview
  - Write invalidation: before a write to a page is allowed, all other read-only copies are invalidated
  - Strict consistency: a reader always sees the latest value written

Write/read sequence

- **Write sequence**
  - Processor ‘i’ has *write fault* to page ‘p’
  - Processor ‘i’ finds owner of page ‘p’ and sends request
  - Owner of ‘p’ sends ***page*** and its ***copyset*** to ‘i’ and marks ‘p’ entry in its page table ‘nil’ (*copyset = list of processors containing read-only copy of page*)
  - Processor ‘i’ sends **invalidation** messages to all processors in *copyset*

- **Read sequence**
  - Processor ‘i’ has *read fault* to page ‘p’
  - Processor ‘i’ finds owner of page ‘p’
  - Owner of ‘p’ sends copy of ***page*** to ‘i’ and adds ‘i’ to *copyset* of ‘p’. Processor ‘i’ has read-only access to ‘p’

Algorithms used for implementing actions for ‘Read’ and ‘Write’ actions

- Centralized manager scheme
  - Central manager resides on single processor: maintains all data **ownership information**
  - On page fault, processor ‘i’ requests copy of page from central manager
  - Central manager sends request to page owner. If ‘Write’ requested, updates owner information to indicate ‘i’ is the new owner

  - Owner sends copy of page to processor ‘i’ and
    - If ‘Write’, also sends *copyset* of page
    - If ‘Read’, adds ‘i’ to the *copyset* of page
  - On write, central manager sends invalidation messages to all processors in *copyset*

  - Performance issues
    - Two messages are required to locate page owner
    - On ‘Writes’, invalidation messages are sent to all processors in *copyset*
    - Centralized manager can become bottleneck

Algorithms used for implementing actions for ‘Read’ and ‘Write’ actions *(cont.)*

- The fixed distributed manager scheme
  - Distributes the central manager’s role to every processor in the system
  - Every processor keeps track of the owners of a predetermined set of pages (determined by a **mapping function *H***)
  - When a processor ‘i’ faults on page ‘p’, processor ‘i’ contacts processor ***H(p)*** for a copy of the page
  - The rest of the protocol is the same as the one with the centralized manager

- *Note:* In both the centralized and fixed distributed manager schemes, if two or more concurrent accesses to the same page are requested, the requests are serialized by the manager

Algorithms used for implementing actions for 'Read' and 'Write' actions *(cont.)*

- The dynamic distributed manager scheme
  - Every host keeps track of the ownership of the pages that are in its *local* page table
    - Every page table has a field called ***probowner*** (**probable owner**)
    - Initially, ***probowner*** is set to a default processor
    - The field is modified as pages are requested from various processors
  - When a processor has a *page fault*, it sends a page request to processor 'i' indicated by the ***probowner*** field
  - If processor 'i' is the true owner of the page, fault handling proceeds like in centralized scheme
  - If 'i' is **not** the owner, it ***forwards*** the request to the processor indicated in its ***probowner*** field
  - This continues until the true owner of the page is found

GiantVM의 주요 소스 코드

QEMU

kvm_cpu_exec(CPUState *cpu)

--> kvm_vcpu_ioctl(cpu, KVM_RUN, 0);

KVM

```
kvm_vcpu_ioctl(struct file *filp, unsigned int ioctl, unsigned long arg) {
        switch(ioctl) {
            case KVM_RUN:
                ....
                kvm_arch_vcpu_ioctl_run(vcpu, vcpu->run);
                        -> vcpu_run(vcpu);
```

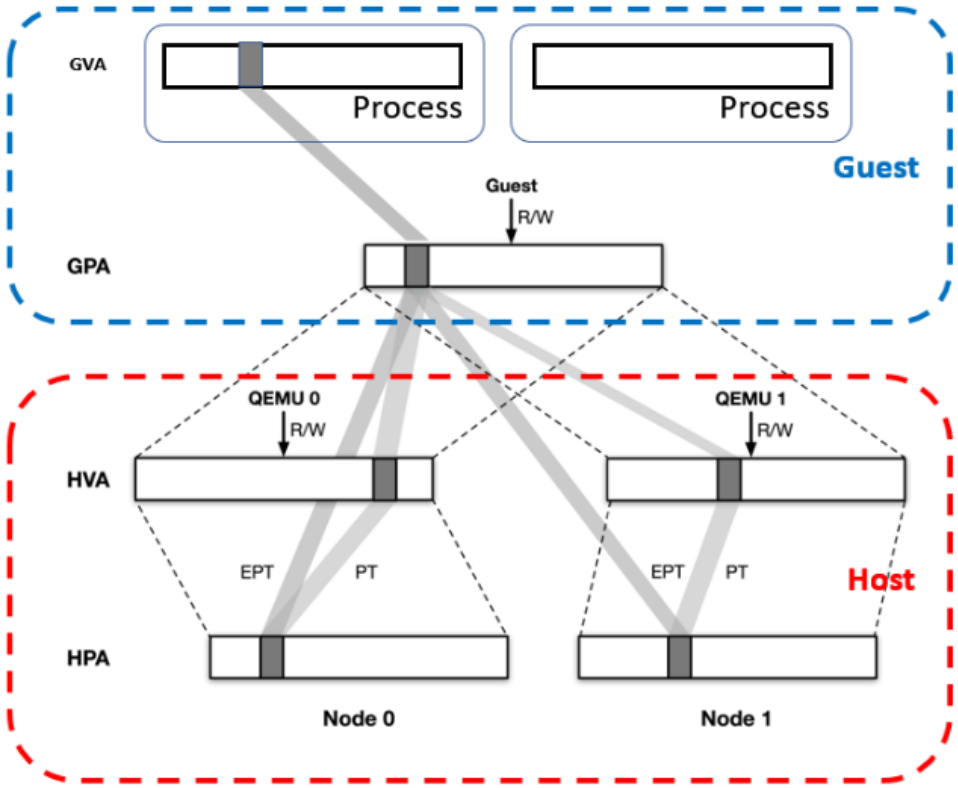
```
static int vcpu_run(struct kvm_vcpu *vcpu) {
        ...
        for(;;) {
            if (kvm_vcpu_running(vcpu)) {
                r = vcpu_enter_guest(vcpu);
            } else {
                r = vcpu_block(kvm, vcpu);
            }
```

```
static int vcpu_enter_guest(struct kvm_vcpu *vcpu) {
        ...
        kvm_x86_ops->prepare_guest_switch(vcpu);
        vcpu->mode = IN_GUEST_MODE;

        kvm_x86_ops->run(vcpu); // → vmx_vcpu_run


        vcpu->mode = OUTSIDE_GUEST_MODE;

        r = kvm_x86_ops->handle_exit(vcpu); // → vmx_handle_exit
```

vcpu_vmx 스트럭처

```
static struct kvm_x86_ops vmx_x86_ops = {
            ...
            .vcpu_create = vmx_create_vcpu,
            .run = vmx_vcpu_run,
            .handle_exit = vmx_handle_exit,
            ...
}
```

EPT violation



GVA → GPA (guest page table)
GPA → HVA (memslot)
HVA → HPA (QEMU page table)
GPA → HPA (EPT)

- 게스트가 처음 페이지를 방문하면 GVA 대 GPA 매핑이 없기 때문에 게스트 OS의 페이지 폴트가 트리거됨
- 그런 다음 게스트 OS는 해당 pte를 설정하고 모든 수준의 페이지 테이블을 복구하고 마지막으로 해당 GPA에 액세스
- HVA에 대한 GPA 매핑이 없기 때문에 EPT violation, KVM에 대한 VMEXIT가 트리거됨
- KVM은 vmx_handle_exit에서 kvm_vmx_exit_handlers[exit_reason]을 실행하고 exit_reason이 EXIT_REASON_EPT_VIOLATION임을 확인하여 *handle_ept_violation*을 호출함

```
vmx_handle_exit(sturct kvm_vcpu *vcpu)
        u32 exit_reason = vmx->exit_reason;
        ...
        return kvm_vmx_exit_handlers[exit_reason](vcpu);
```

```
static int (*const kvm_vmx_exit_handlers[])(struct kvm_vcpu *vcpu) = {
    ....
     [EXIT_REASON_EPT_VIOLATION]        = handle_ept_violation,
```

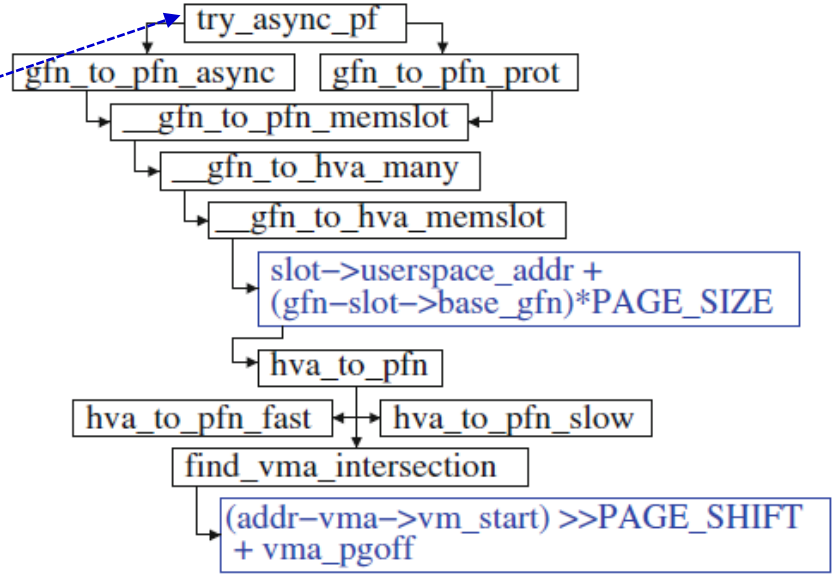
```
static int handle_ept_violation(struct kvm_vcpu *vcpu)
        ...
        exit_qualification = vmcs_readl(EXIT_QUALIFICATION);
        gpa = vmcs_read64(GUEST_PHYSICAL_ADDRESS);
        /* it is a read fault? */
        error_code = (exit_qualification << 2) & PFERR_USER_MASK;
        /* it is a write fault? */
        error_code |= exit_qualification & PFERR_WRITE_MASK;
        /* It is a fetch fault? */
        error_code |= (exit_qualification << 2) & PFERR_FETCH_MASK;
        /* ept page table is present? */
         error_code |= (exit_qualification & 0x38) != 0;
        return kvm_mmu_page_fault(vcpu, gpa, error_code, NULL, 0);
```

```
int kvm_mmu_page_fault(struct kvm_vcpu *vcpu, gva_t cr2, u32 error_code,
                void *insn, int insn_len)
    ....
    r = vcpu->arch.mmu.page_fault(vcpu, cr2, error_code, false);
```

```
static void init_kvm_tdp_mmu(struct kvm_vcpu *vcpu)
{
    struct kvm_mmu *context = &vcpu->arch.mmu;

    context->base_role.word = 0;
    context->base_role.smm = is_smm(vcpu);
    context->page_fault = tdp_page_fault;
    context->sync_page = nonpaging_sync_page;
```

(1) Host page frame number calculation for faulting address

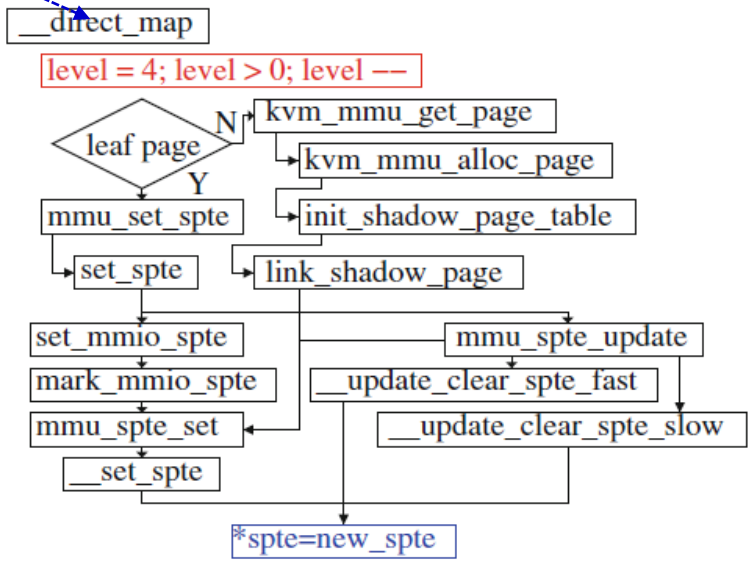
```
static int tdp_page_fault(struct kvm_vcpu *vcpu, gva_t gpa, u32 error_code,
                bool prefault)
        ...
        if (try_async_pf(vcpu, prefault, gfn, gpa, &pfn, write, &map_writable))
            return 0;

        dsm_access = kvm_dsm_vcpu_acquire_page(vcpu, &slot, gfn, write);
        ...
        r = __direct_map(vcpu, write, map_writable, level, gfn, pfn, prefault, dsm_access);
```



--- DSM 핵심 코드 ---

(2) Acquire page on DSM (뒷장 참조)

(3) mapping into TDP tables



[note]
KVM에서는 SPT와 TDP가 동일한 자료 구조를 공유 (똑같이, shadow page table이라고 명기)

(2) Acquire page on DSM (앞장에 이어서)

```
int kvm_dsm_vcpu_acquire_page(struct kvm_vcpu *vcpu,
        struct kvm_memory_slot **slot, gfn_t gfn, bo
{
    struct kvm_memory_slot *memslot;
    memslot = kvm_vcpu_gfn_to_memslot(vcpu, gfn);
    if (slot)
        *slot = memslot;
    return __kvm_dsm_acquire_page(vcpu->kvm, memslot
            gfn, is_smm(vcpu), write);
}
```

```
static int __kvm_dsm_acquire_page(struct kvm *kvm,
        struct kvm_memory_slot *slot, gfn_t gfn, bool is_smm, bool write)
{
    struct kvm_dsm_memory_slot *hvaslot;
    hfn_t vfn;
    int dsm_access;

    if (WARN_ON(kvm->mm != current->mm))
        return -EINVAL;
    if (!kvm->arch.dsm_enabled)
        return ACC_ALL;

    /*
     * We should ignore private memslots since they are not really visible
     * to guest and thus are not part of guest state that should be
     * distributedly shared.
     */
    if (!slot || slot->id >= KVM_USER_MEM_SLOTS ||
            slot->flags & KVM_MEMSLOT_INVALID)
        return ACC_ALL;

    vfn = __gfn_to_vfn_memslot(slot, gfn);
    hvaslot = gfn_to_hvaslot(kvm, slot, gfn);
    if (!hvaslot)
        return ACC_ALL;

    dsm_lock(kvm, hvaslot, vfn);
    dsm_access = kvm_dsm_page_fault(kvm, slot, gfn, is_smm, write);
    if (dsm_access < 0) {
        dsm_unlock(kvm, hvaslot, vfn);
    }
    return dsm_access;
```

```
static int kvm_dsm_page_fault(struct kvm *kvm, struct kvm_memory_slot *memslot,
        gfn_t gfn, bool is_smm, int write)
{
    int ret;
#ifdef KVM_DSM_PF_PROFILE
    struct timespec ts;
    ulong start;

    getnstimeofday(&ts);
    start = ts.tv_sec * 1000 * 1000 + ts.tv_nsec / 1000;
#endif

#ifdef IVY_KVM_DSM
    ret = ivy_kvm_dsm_page_fault(kvm, memslot, gfn, is_smm, write);
#elif defined(TARDIS_KVM_DSM)
```

```
if (dsm_is_owner(slot, vfn)) {
    BUG_ON(dsm_get_prob_owner(slot, vfn) != kvm->arch.dsm_id);

    ret = kvm_dsm_invalidate(kvm, gfn, is_smm, slot, vfn, NULL, kvm->arch.dsm_id);
    if (ret < 0)
        goto out_error;
    resp.version = dsm_get_version(slot, vfn);
    resp_len = PAGE_SIZE;

    dsm_incr_version(slot, vfn);
}
else {
    owner = dsm_get_prob_owner(slot, vfn);
    /* Owner of all pages is 0 on init. */
    if (unlikely(dsm_is_initial(slot, vfn) && kvm->arch.dsm_id == 0)) {
        dsm_set_prob_owner(slot, vfn, kvm->arch.dsm_id);
        dsm_change_state(slot, vfn, DSM_OWNER | DSM_MODIFIED);
        dsm_add_to_copyset(slot, vfn, kvm->arch.dsm_id);
        ret = ACC_ALL;
        goto out;
    }
    /*
     * Ask the probOwner. The prob(ably) owner is probably true owner,
     * or not. If not, forward the request to next probOwner until find
     * the true owner.
     */
    ret = resp_len = kvm_dsm_fetch(kvm, owner, false, &req, page,
            &resp);
    if (ret < 0)
        goto out_error;
    ret = kvm_dsm_invalidate(kvm, gfn, is_smm, slot, vfn,
        &resp.inv_copyset, owner);
    if (ret < 0)
        goto out_error;

    dsm_set_version(slot, vfn, resp.version + 1);
}

dsm_clear_copyset(slot, vfn);
dsm_add_to_copyset(slot, vfn, kvm->arch.dsm_id);

dsm_decode_diff(page, resp_len, memslot, gfn);
dsm_set_twin_conditionally(slot, vfn, page, memslot, gfn,
        dsm_is_owner(slot, vfn), resp.version);

if (!dsm_is_owner(slot, vfn) && resp_len > 0) {
    ret = __kvm_write_guest_page(memslot, gfn, page, 0, PAGE_SIZE);
    if (ret < 0) {
        goto out_error;
    }
}

dsm_set_prob_owner(slot, vfn, kvm->arch.dsm_id);
dsm_change_state(slot, vfn, DSM_OWNER | DSM_MODIFIED);
```

ivy_kvm_dsm_page_fault (→ write fault)

```
struct dsm_request req = {
    .req_type = DSM_REQ_WRITE,
    .requester = kvm->arch.dsm_id,
    .msg_sender = kvm->arch.dsm_id,
    .gfn = gfn,
    .is_smm = is_smm,
    .version = dsm_get_version(slot, vfn),
};
```

(1) Probable owner 를 찾음

(2) Probable owner에게 request 를 전송

(request를 전송받은 probable owner가 진짜 owner가 아니라면, 찾을 때까지 forwarding. 인출된 page는 파이프라인형태로 전달됨)

(3) Owner로부터 page와 기존 copyset을 전달받음

(original owner는 INVALID 상태로 전환 → owner 지위를 잃음 & copyset에서 original owner 비트를 빼고 전달)

(4) Copyset에 포함된 모든 노드들에게 invalidation 메시지를 전송

(5) Copyset을 clear하고, 자신(현 write fault 노드)를 copyset에 등록

```
int __kvm_write_guest_page(struct kvm_memory_slot *memslot, gfn_t gfn,
                           const void *data, int offset, int len)
{
    int r;
    unsigned long addr;

    addr = gfn_to_hva_memslot(memslot, gfn);
    if (kvm_is_error_hva(addr))
        return -EFAULT;
    r = __copy_to_user((void __user *)addr + offset, data, len);
    if (r)
        return -EFAULT;
    mark_page_dirty_in_slot(memslot, gfn);
    return 0;
}
```

(6) (앞서 원격 copy해온) *page*를 *hva 영역으로* 다시 copy

page

(7) 자신을 owner로 설정

ivy_kvm_dsm_page_fault (→ read fault)

```
struct dsm_request req = {
    .req_type = DSM_REQ_READ,
    .requester = kvm->arch.dsm_id,
    .msg_sender = kvm->arch.dsm_id,
    .gfn = gfn,
    .is_smm = is_smm,
    .version = dsm_get_version(slot, vfn),
};
owner = dsm_get_prob_owner(slot, vfn);
/*
 * If I'm the owner, then I would have already been in Shared or
 * Modified state.
 */
BUG_ON(dsm_is_owner(slot, vfn));

/* Owner of all pages is 0 on init. */
if (unlikely(dsm_is_initial(slot, vfn) && kvm->arch.dsm_id == 0)) {
    dsm_set_prob_owner(slot, vfn, kvm->arch.dsm_id);
    dsm_change_state(slot, vfn, DSM_OWNER | DSM_SHARED);
    dsm_add_to_copyset(slot, vfn, kvm->arch.dsm_id);
    ret = ACC_EXEC_MASK | ACC_USER_MASK;
    goto out;
}
/* Ask the probOwner */
ret = resp_len = kvm_dsm_fetch(kvm, owner, false, &req, page, &resp);
if (ret < 0)
    goto out_error;

dsm_set_version(slot, vfn, resp.version);
memcpy(dsm_get_copyset(slot, vfn), &resp.inv_copyset, sizeof(copyset_t
dsm_add_to_copyset(slot, vfn, kvm->arch.dsm_id);

dsm_decode_diff(page, resp_len, memslot, gfn);

ret = __kvm_write_guest_page(memslot, gfn, page, 0, PAGE_SIZE);
if (ret < 0)
    goto out_error;

dsm_set_prob_owner(slot, vfn, kvm->arch.dsm_id);
dsm_change_state(slot, vfn, DSM_OWNER | DSM_SHARED);
ret = ACC_EXEC_MASK | ACC_USER_MASK;
```

(1) Probable owner 를 찾음

(2) Probable owner에게 request 를 전송

(request를 전송받은 probable owner가 진짜 owner가 아니라면, 찾을 때까지 forwarding.
인출된 page는 파이프라인형태로 전달됨)

(3) Owner로부터 page와 *copyset을* 전달받음

(original owner는 SHARED 상태로 전환 → owner 지위를 잃음
& original owner 노드의 prob owner는 요청자(현 read fault 노드)로 설정됨)

(4) *copyset*에 자신을 추가

```
int __kvm_write_guest_page(struct kvm_memory_slot *memslot, gfn_t gfn,
                           const void *data, int offset, int len)
{
    int r;
    unsigned long addr;

    addr = gfn_to_hva_memslot(memslot, gfn);
    if (kvm_is_error_hva(addr))
        return -EFAULT;
    r = __copy_to_user((void __user *)addr + offset, data, len);
    if (r)
        return -EFAULT;
    mark_page_dirty_in_slot(memslot, gfn);
    return 0;
}
```

(5) (앞서 copy해온) *page*를 *hva 영역으로* 다시 copy

page

(6) 자신을 owner로 설정

특이사항: read인데도. *original owner node가 권한을 뺏김!*

GiantVM DSM의 특기 사항 일부

#1) write fault시, local 처리 가능성

```
owner = dsm_get_prob_owner(slot, vfn);
/* Owner of all pages is 0 on init. */
if (unlikely(dsm_is_initial(slot, vfn) && kvm->arch.dsm_id == 0)) {
    dsm_set_prob_owner(slot, vfn, kvm->arch.dsm_id);
    dsm_change_state(slot, vfn, DSM_OWNER | DSM_MODIFIED);
    dsm_add_to_copyset(slot, vfn, kvm->arch.dsm_id);
    ret = ACC_ALL;
    goto out;
}
/*
 * Ask the probOwner. The prob(ably) owner is probably true owner,
 * or not. If not, forward the request to next probOwner until find
 * the true owner.
 */
ret = resp_len = kvm_dsm_fetch(kvm, owner, false, &req, page,
        &resp);
if (ret < 0)
    goto out_error;
ret = kvm_dsm_invalidate(kvm, gfn, is_smm, slot, vfn,
        &resp.inv_copyset, owner);
if (ret < 0)
    goto out_error;

dsm_set_version(slot, vfn, resp.version + 1);
```

메모리 슬롯의 state 값이 DSM_INITIAL(== 0) 이면, 네트웍으로 fetch해서 page를 가져오는 대신, local page를 활용

- 갓 기동된 VM에서 최초 write fault 시 네트웍 타지 않고 local page 활용이 가능

- state에 0을 설정함으로써, DSM 수준에서 local page 사용의 힌트를 줄 수 있어 보임 (그러면, 언제 0을 설정할 것인가?)

```
static inline bool dsm_is_initial(struct kvm_dsm_memory_slot *slot, hfn_t vfn)
{
    return (slot->vfn_dsm_state[vfn - slot->base_vfn].state &
            DSM_MSI_STATE_MASK) == DSM_INITIAL;
}
```

0

```
#define DSM_INITIAL     0
#define DSM_INVALID     1
#define DSM_SHARED      2
#define DSM_MODIFIED    3
```

```
int dsm_create_memslot(struct kvm_dsm_memory_slot *slot,
        unsigned long npages)
{
    unsigned long i;
    int ret = 0;

    slot->vfn_dsm_state = kvm_kvzalloc(npages * sizeof(*slot->vfn_dsm_state));
    if (!slot->vfn_dsm_state)
        return -ENOMEM;
```

vfn_dsm_state 는 메모리슬롯 생성시 0으로 fill됨 (*kvm_kvzalloc → __GFP_ZERO 옵션 사용*)

#2) read 요청 측이 새로운 owner가 됨 (← original ivy protocol과 상이함)

Read 요청 측 (read page fault)

```
owner = dsm_get_prob_owner(slot, vfn);
/*
 * If I'm the owner, then I would have already been in Shared or
 * Modified state.
 */
BUG_ON(dsm_is_owner(slot, vfn));

/* Owner of all pages is 0 on init. */
if (unlikely(dsm_is_initial(slot, vfn) && kvm->arch.dsm_id == 0)) {
    dsm_set_prob_owner(slot, vfn, kvm->arch.dsm_id);
    dsm_change_state(slot, vfn, DSM_OWNER | DSM_SHARED);
    dsm_add_to_copyset(slot, vfn, kvm->arch.dsm_id);
    ret = ACC_EXEC_MASK | ACC_USER_MASK;
    goto out;
}
/* Ask the probOwner */
ret = resp_len = kvm_dsm_fetch(kvm, owner, false, &req, page, &resp);
if (ret < 0)
    goto out_error;
```

전송받은 copyset에 자신을 추가 → 신규 copyset 설정

```
dsm_set_version(slot, vfn, resp.version);
memcpy(dsm_get_copyset(slot, vfn), &resp.inv_copyset, sizeof(copyset_t));
dsm_add_to_copyset(slot, vfn, kvm->arch.dsm_id);

dsm_decode_diff(page, resp_len, memslot, gfn);
```

전송받은 *page*를 hva 영역에 복사. *page*는 버림

```
ret = __kvm_write_guest_page(memslot, gfn, page, 0, PAGE_SIZE);
if (ret < 0)
    goto out_error;
```

prob owner에 자기 자신을 설정

```
dsm_set_prob_owner(slot, vfn, kvm->arch.dsm_id);
```

자기 자신을 OWNER로 설정

```
dsm_change_state(slot, vfn, DSM_OWNER | DSM_SHARED);
```

서버 로직(at original owner)

```
if ((is_owner = dsm_is_owner(slot, vfn))) {
    BUG_ON(dsm_get_prob_owner(slot, vfn) != kvm->arch.dsm_id);
```

Prob owenr에 read 요청측을 설정

```
    dsm_set_prob_owner(slot, vfn, req->msg_sender);
    dsm_debug_v("kvm[%d](S1) changed owner of gfn[%llu,%d] "
            "from kvm[%d] to kvm[%d]\n", kvm->arch.dsm_id, req->gfn,
            req->is_smm, kvm->arch.dsm_id, req->msg_sender);
    /* TODO: if modified */
```

OWNER 표식을 삭제

```
    dsm_change_state(slot, vfn, DSM_SHARED);
    kvm_dsm_apply_access_right(kvm, slot, vfn, DSM_SHARED);

    ret = kvm_read_guest_page_nonlocal(kvm, memslot, req->gfn, page, 0, PAGE_SIZE);
    if (ret < 0)
        goto out;
```

기존 page를 전송 준비

```
    /*
     * read fault causes owner transmission, too. Send copyset back to new
     * owner.
     */
    resp.inv_copyset = *dsm_get_copyset(slot, vfn);
    BUG_ON(!(test_bit(kvm->arch.dsm_id, &resp.inv_copyset)));
    resp.version = dsm_get_version(slot, vfn);
}
else if (dsm_is_initial(slot, vfn) && kvm->arch.dsm_id == 0) {
```

기존 copyset을 전송 준비

- *Original ivy 프로토콜에서는 read fault 시 original owner 가 owner 역할을 유지*
  (→ 그러나, GiantVM에서는 read fault측이 owner로 변경됨)

#3) 최초 초기화에서 non-master 노드의 비효율

write page fault

```
owner = dsm_get_prob_owner(slot, vfn);
/* Owner of all pages is 0 on init. */
if (unlikely(dsm_is_initial(slot, vfn) && kvm->arch.dsm_id == 0)) {
    dsm_set_prob_owner(slot, vfn, kvm->arch.dsm_id);
    dsm_change_state(slot, vfn, DSM_OWNER | DSM_MODIFIED);
    dsm_add_to_copyset(slot, vfn, kvm->arch.dsm_id);
    ret = ACC_ALL;
    goto out;
}
/*
 * Ask the probOwner. The prob(ably) owner is probably true owner,
 * or not. If not, forward the request to next probOwner until find
 * the true owner.
 */
ret = resp_len = kvm_dsm_fetch(kvm, owner, false, &req, page,
        &resp);
if (ret < 0)
    goto out_error;
ret = kvm_dsm_invalidate(kvm, gfn, is_smm, slot, vfn,
        &resp.inv_copyset, owner);
if (ret < 0)
    goto out_error;
dsm_set_version(slot, vfn, resp.version + 1);
```

Master 에서는 locally 처리

Non-master 에서는 remote 요청-처리

(참고) non-master 는 kvm->arch.dsm_id != 0

서버 로직(at master)

```
if ((is_owner = dsm_is_owner(slot, vfn))) {
    BUG_ON(dsm_get_prob_owner(slot, vfn) != kvm->arch.dsm_id);

    /* I'm owner */
    dsm_set_prob_owner(slot, vfn, req->msg_sender);
    dsm_debug_v("kvm[%d](M1) changed owner of gfn[%llu,%d] "
            "from kvm[%d] to kvm[%d]\n", kvm->arch.dsm_id, req->gfn,
            req->is_smm, kvm->arch.dsm_id, req->msg_sender);
    dsm_change_state(slot, vfn, DSM_INVALID);
    kvm_dsm_apply_access_right(kvm, slot, vfn, DSM_INVALID);
    /* Send back copyset to new owner. */
    resp.inv_copyset = *dsm_get_copyset(slot, vfn);
    resp.version = dsm_get_version(slot, vfn);
    clear_bit(kvm->arch.dsm_id, &resp.inv_copyset);
    ret = kvm_read_guest_page_nonlocal(kvm, memslot, req->gfn, page, 0, PAGE_SIZE);
    if (ret < 0)
        return ret;
}
else if (dsm_is_initial(slot, vfn) && kvm->arch.dsm_id == 0) {
    /* Send back a dummy copyset. */
    resp.inv_copyset = 0;
    resp.version = dsm_get_version(slot, vfn);
    ret = kvm_read_guest_page_nonlocal(kvm, memslot, req->gfn, page, 0, PAGE_SIZE);
    if (ret < 0)
        return ret;
    dsm_set_prob_owner(slot, vfn, req->msg_sender);
    dsm_change_state(slot, vfn, DSM_INVALID);
}
else {
```

Non-master node의 최초 write용 page는 master node에게 요청해서 dummy page를 받아와서 사용
(최소한, master에게 prob owner 정보만 보내고, locally 할당할 최적화 여지가 있음)