Rmap -3- (PVMW)

2019-09-10 (http://jake.dothome.co.kr/rmap-3/) Moon Young-il (http://jake.dothome.co.kr/author/admin/) Linux Kernel (http://jake.dothome.co.kr/category/linux/)

<kernel v5.0>

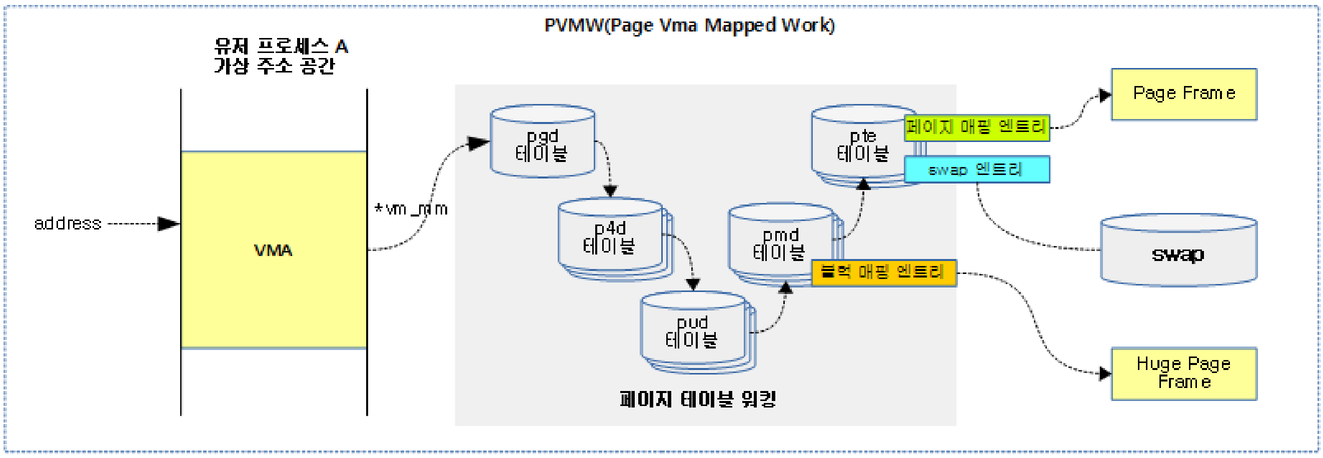
Rmap -3- (PVMW)

PVMW(Page Vma Mapped Walk)

This is the interface to check whether a page has been mapped to the VMA.

- Note: mm: introduce page_vma_mapped_walk() (https://github.com/torvalds/linux/commit/ace71a19cec5eb430207c3269d8a2683f0574306#diff-82d4b79a51478b4ef923b8d2f2ffdee6)

The following figure shows whether it is a physical page mapping that corresponds to a virtual address or a mapping to a swap entry.



(http://jake.dothome.co.kr/wp-content/uploads/2019/09/pvmw-1.png)

Request Flags

The flags used in PVMW and the functions used are as follows:

- PVMW_SYNC
  - map_pte() function, which allows it to bypass strict check routines.
  - page_mkclean_one()
  - page_mapped_in_vma()
  - remove_migration_pte()

- PVMW_MIGRATION
  - migration.
  - remove_migration_pte()
- NO Flag
  - try_to_unmap_one()
  - page_referenced_one()
  - __replace_page()
  - page_idle_clear_pte_refs_one()
  - write_protect_page()

Verify that a page is VMA mapped

page_vma_mapped_walk()

mm/page_vma_mapped.c -1/2-

```
/**
 * page_vma_mapped_walk - check if @pvmw->page is mapped in @pvmw->vma at
 * @pvmw->address
 * @pvmw: pointer to struct page_vma_mapped_walk. page, vma, address and flags
 * must be set. pmd, pte and ptl must be NULL.
 *
 * Returns true if the page is mapped in the vma. @pvmw->pmd and @pvmw->pte point
 * to relevant page table entries. @pvmw->ptl is locked. @pvmw->address is
 * adjusted if needed (for PTE-mapped THPs).
 *
 * If @pvmw->pmd is set but @pvmw->pte is not, you have found PMD-mapped page
 * (usually THP). For PTE-mapped THP, you should run page_vma_mapped_walk() in
 * a loop to find all PTEs that map the THP.
 *
 * For HugeTLB pages, @pvmw->pte is set to the relevant page table entry
 * regardless of which page table level the page is mapped at. @pvmw->pmd is
 * NULL.
 *
 * Retruns false if there are no more page table entries for the page in
 * the vma. @pvmw->ptl is unlocked and @pvmw->pte is unmapped.
 *
 * If you need to stop the walk before page_vma_mapped_walk() returned false,
 * use page_vma_mapped_walk_done(). It will do the housekeeping.
 */
```

```
bool page_vma_mapped_walk(struct page_vma_mapped_walk *pvmw)
{
        struct mm_struct *mm = pvmw->vma->vm_mm;
        struct page *page = pvmw->page;
        pgd_t *pgd;
        p4d_t *p4d;
        pud_t *pud;
        pmd_t pmde;

        /* The only possible pmd mapping has been handled on last iteration */
        if (pvmw->pmd && !pvmw->pte)
                return not_found(pvmw);

        if (pvmw->pte)
                goto next_pte;

        if (unlikely(PageHuge(pvmw->page))) {
                /* when pud is not present, pte will be NULL */
                pvmw->pte = huge_pte_offset(mm, pvmw->address,
                                            PAGE_SIZE << compound_order(page));
                if (!pvmw->pte)
                        return false;

                pvmw->ptl = huge_pte_lockptr(page_hstate(page), mm, pvmw->pte);
                spin_lock(pvmw->ptl);
                if (!check_pte(pvmw))
                        return not_found(pvmw);
                return true;
        }
restart:
        pgd = pgd_offset(mm, pvmw->address);
        if (!pgd_present(*pgd))
                return false;
        p4d = p4d_offset(pgd, pvmw->address);
        if (!p4d_present(*p4d))
                return false;
        pud = pud_offset(p4d, pvmw->address);
        if (!pud_present(*pud))
                return false;
        pvmw->pmd = pmd_offset(pud, pvmw->address);
        /*
         * Make sure the pmd value isn't cached in a register by the
         * compiler and used as a stale value after we've observed a
         * subsequent update.
         */
        pmde = READ_ONCE(*pvmw->pmd);
        if (pmd_trans_huge(pmde) || is_pmd_migration_entry(pmde)) {
                pvmw->ptl = pmd_lock(mm, pvmw->pmd);
                if (likely(pmd_trans_huge(*pvmw->pmd))) {
                        if (pvmw->flags & PVMW_MIGRATION)
                                return not_found(pvmw);
                        if (pmd_page(*pvmw->pmd) != page)
                                return not_found(pvmw);
                        return true;
                } else if (!pmd_present(*pvmw->pmd)) {
                        if (thp_migration_supported()) {
                                if (!(pvmw->flags & PVMW_MIGRATION))
                                        return not_found(pvmw);
                                if (is_migration_entry(pmd_to_swp_entry(*pvmw->pmd))) {
                                        swp_entry_t entry = pmd_to_swp_entry(*pvmw->pmd);

                                        if (migration_entry_to_page(entry) != page)
                                                return not_found(pvmw);
                                        return true;
                                }
                        }
                        return not_found(pvmw);
                } else {
                        /* THP pmd was split under us: handle on pte level */
                        spin_unlock(pvmw->ptl);
                        pvmw->ptl = NULL;
                }
73          } else if (!pmd_present(pmde)) {
74                  return false;
75          }
76          if (!map_pte(pvmw))
77                  goto next_pte;
```

Returns whether @pvmw->page is already mapped to @pvmw->vma.

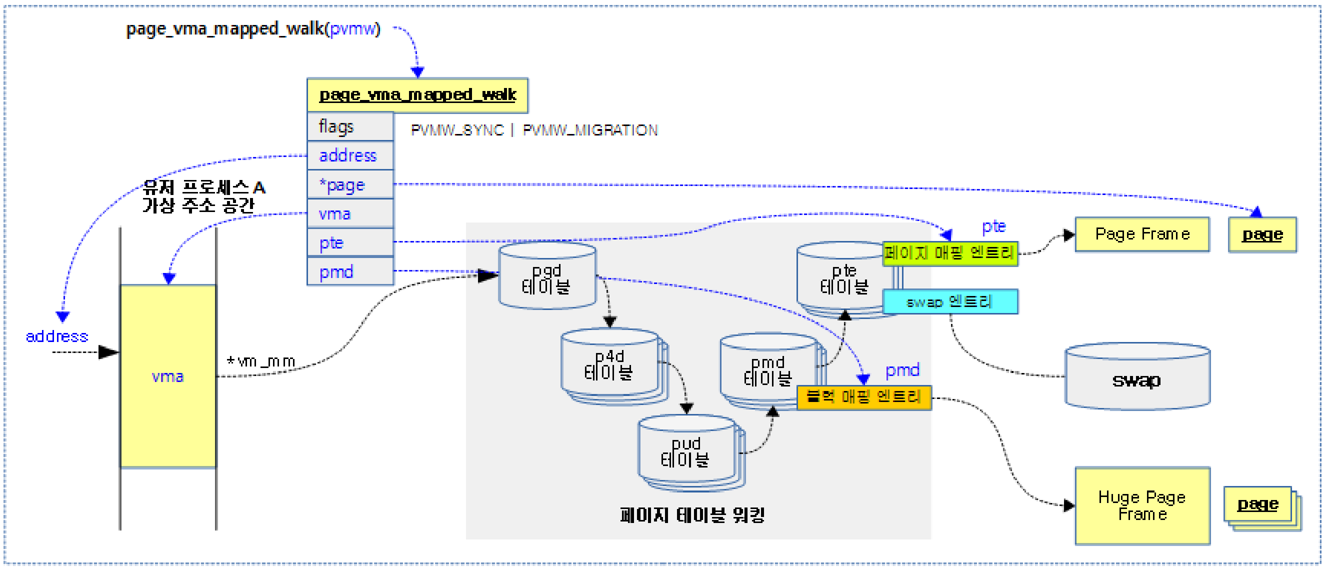
- In line 11~12 of the code, if only pmd has a mapping and no pte, it completes the page_vma_mappled_walk and returns false.
- If there is a PTE mapping in line 14~15 of the code, go directly to the next_pte label.
- In line 17~29 of the code, if it is a huge page with a low probability, it will exit the function after specifying the pte.
- On line 30~40 of the code, the restart: label is. pgd -> p4d -> pud -> pmd entries. However, if there are no entries, it returns false.
- Code line 46~72 This is the treatment for cases where the pmd entry is a trans huge or migration entry. Returns true if the steps up to the pmd entry are complete, and fasle if the request is otherwise or inappropriate. However, if it is splitting, release the PTL lock and continue.
- On lines 73~75 of the code, it returns false if the pmd entry does not exist.
- In code lines 76~77, if the PTE entry is not mapped, go to the next_pte label.

mm/page_vma_mapped.c -2/2-

```
        while (1) {
                if (check_pte(pvmw))
                        return true;
next_pte:
                /* Seek to next pte only makes sense for THP */
                if (!PageTransHuge(pvmw->page) || PageHuge(pvmw->page))
                        return not_found(pvmw);
                do {
                        pvmw->address += PAGE_SIZE;
                        if (pvmw->address >= pvmw->vma->vm_end ||
                            pvmw->address >=
                                        __vma_address(pvmw->page, pvmw->vma) +
                                        hpage_nr_pages(pvmw->page) * PAGE_SIZE)
                                return not_found(pvmw);
                        /* Did we cross page table boundary? */
                        if (pvmw->address % PMD_SIZE == 0) {
                                pte_unmap(pvmw->pte);
                                if (pvmw->ptl) {
                                        spin_unlock(pvmw->ptl);
                                        pvmw->ptl = NULL;
                                }
                                goto restart;
                        } else {
                                pvmw->pte++;
                        }
                } while (pte_none(*pvmw->pte));

                if (!pvmw->ptl) {
                        pvmw->ptl = pte_lockptr(mm, pvmw->pmd);
                        spin_lock(pvmw->ptl);
                }
        }
}
```

- In line 1~3 of the code, it traverses the pte entries, strictly checking that each page is mapped, and returns true if it does.
- In code lines 4~7, next_pte: Label. If it's not thp or huge, complete the page_vma_mappled_walk and return false.
- Repeat if there are no PTE entries mapped in code lines 8~26, complete page_vma_mappled_walk if the address is outside the VMA zone range, and return false.
- If the PTL lock is released in code lines 28~31, acquire the lock again and repeat.

The following figure shows how the page_vma_mapped_walk() function is used to determine whether the address has been mapped correctly.



(http://jake.dothome.co.kr/wp-content/uploads/2019/09/page_vma_mapped_walk-1.png)

check_pte()

mm/page_vma_mapped.c

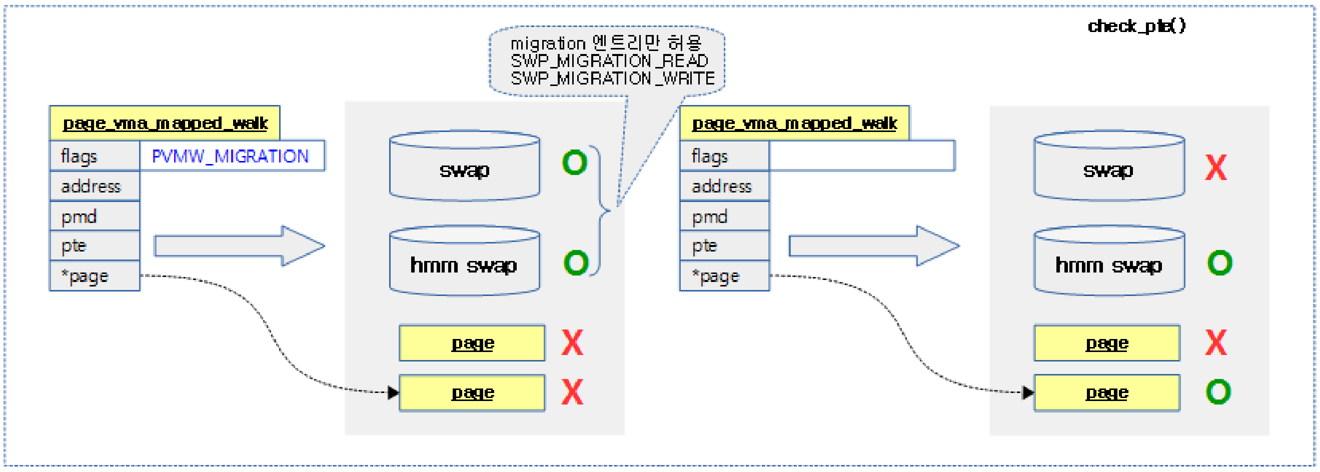
```
/**
 * check_pte - check if @pvmw->page is mapped at the @pvmw->pte
 *
 * page_vma_mapped_walk() found a place where @pvmw->page is *potentially*
 * mapped. check_pte() has to validate this.
 *
 * @pvmw->pte may point to empty PTE, swap PTE or PTE pointing to arbitrary
 * page.
 *
 * If PVMW_MIGRATION flag is set, returns true if @pvmw->pte contains migration
 * entry that points to @pvmw->page or any subpage in case of THP.
 *
 * If PVMW_MIGRATION flag is not set, returns true if @pvmw->pte points to
 * @pvmw->page or any subpage in case of THP.
 *
 * Otherwise, return false.
 *
 */
```

```
01 tatic bool check_pte(struct page_vma_mapped_walk *pvmw)
02 {
03         unsigned long pfn;
04 
05         if (pvmw->flags & PVMW_MIGRATION) {
06                 swp_entry_t entry;
07                 if (!is_swap_pte(*pvmw->pte))
08                         return false;
09                 entry = pte_to_swp_entry(*pvmw->pte);
10 
11                 if (!is_migration_entry(entry))
12                         return false;
13 
14                 pfn = migration_entry_to_pfn(entry);
15         } else if (is_swap_pte(*pvmw->pte)) {
16                 swp_entry_t entry;
17 
18                 /* Handle un-addressable ZONE_DEVICE memory */
19                 entry = pte_to_swp_entry(*pvmw->pte);
20                 if (!is_device_private_entry(entry))
21                         return false;
22 
23                 pfn = device_private_entry_to_pfn(entry);
24         } else {
25                 if (!pte_present(*pvmw->pte))
26                         return false;
27 
28                 pfn = pte_pfn(*pvmw->pte);
29         }
30 
31         return pfn_in_hpage(pvmw->page, pfn);
32 }
```

Returns whether the page is mapped. (pvmw->pte entries — > pvmw->page mapping, but whether swap & migration entries are mapped at migration time)

- In code lines 5~14, if there is a PVMW_MIGRATOIN flag request, get the pfn for the swap pte entry. If it is not a swap pte or a migration entry, it returns false.
- In code lines 15~23, it returns false if it is mapped to a swap pte entry, but if the swap pte entry is a device private entry, it knows the corresponding pfn.
    - The swap device is swapped to HMM memory that is not managed by Buddy.
- If the pte entry is mapped from code lines 24~29, the pfn of the page is known.
- Returns whether the pfn generated on line 31 exists within the pfn range of pvmw->page, which is either a regular page or a huge page.

The following figure checks whether the PWMC->pte entry corresponding to PWMC->Address is mapped to PWMC->page.



(http://jake.dothome.co.kr/wp-content/uploads/2019/09/check_pte-1.png)

map_pte()

mm/page_vma_mapped.c

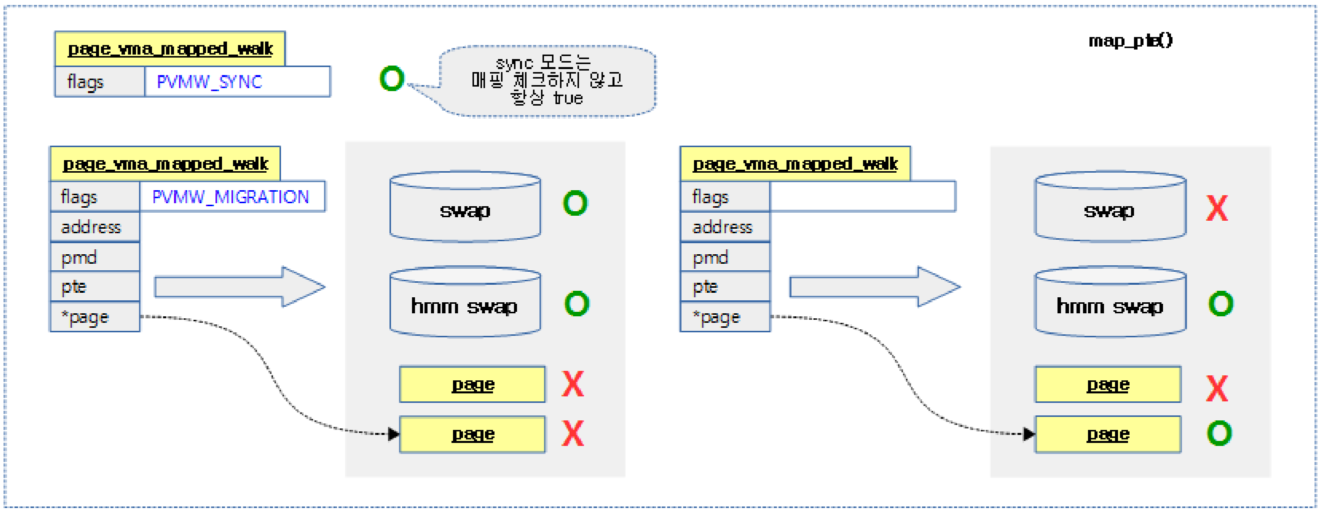
```
static bool map_pte(struct page_vma_mapped_walk *pvmw)
{
        pvmw->pte = pte_offset_map(pvmw->pmd, pvmw->address);
        if (!(pvmw->flags & PVMW_SYNC)) {
                if (pvmw->flags & PVMW_MIGRATION) {
                        if (!is_swap_pte(*pvmw->pte))
                                return false;
                } else {
                        /*
                         * We get here when we are trying to unmap a private
                         * device page from the process address space. Such
                         * page is not CPU accessible and thus is mapped as
                         * a special swap entry, nonetheless it still does
                         * count as a valid regular mapping for the page (and
                         * is accounted as such in page maps count).
                         *
                         * So handle this special case as if it was a normal
                         * page mapping ie lock CPU page table and returns
                         * true.
                         *
                         * For more details on device private memory see HMM
                         * (include/linux/hmm.h or mm/hmm.c).
                         */
                        if (is_swap_pte(*pvmw->pte)) {
                                swp_entry_t entry;

                                /* Handle un-addressable ZONE_DEVICE memory */
                                entry = pte_to_swp_entry(*pvmw->pte);
                                if (!is_device_private_entry(entry))
                                        return false;
                        } else if (!pte_present(*pvmw->pte))
                                return false;
                }
        }
        pvmw->ptl = pte_lockptr(pvmw->vma->vm_mm, pvmw->pmd);
36          spin_lock(pvmw->ptl);
37          return true;
38  }
```

Returns whether the page is mapped. (pvmw->pte entry —> pvmw->page mapping, but whether or not swap entry is mapped on migration) However, in sync mode, it is unchecked and always returns true.

- In line 3 of code, use pvmw->pmd and pvmw->address to get the pte entry.
- If it is requested with the PVMW_SYNC flag in line 4, it returns true without the following strict checks:
- Returns false if requested with PVMW_MIGRATOIN flag in line 5~7 of the code. However, this is not the case when the PTE entry is mapped to a swap_pte.
- In code lines 8~33, if it is mapped to a swap device, it returns false unless it is swapped into HMM memory that is not managed by Buddy. It also returns false if it is not mapped to a swap device and is mapped to a page that does not exist.
- This is the case when the PTE entry in code lines 35~37 is mapped normally. PTL lock and returns true.

The following figure checks whether the PWMC->pte entry corresponding to PWMC->Address is mapped to PWMC->page.

- check_pte(). map_pte() always returns true when requesting sync mode.



(http://jake.dothome.co.kr/wp-content/uploads/2019/09/map_pte-1.png)

page_vma_mapped_walk Structure

include/linux/rmap.h

```
struct page_vma_mapped_walk {
        struct page *page;
        struct vm_area_struct *vma;
        unsigned long address;
        pmd_t *pmd;
        pte_t *pte;
        spinlock_t *ptl;
        unsigned int flags;
};
```

- *page
    - This is a target page to check if it is a mapped page.
- *vma
    - Specify the VMA zone.
- address
    - Virtual Addresses
- pmd
    - A pmd entry with a virtual address mapped to it, which is used for forwarding inside the function.
- pte
    - A virtual address is a mapped PTE entry that is used for forwarding inside the function.
- *ptl
    - PVMW is used inside the function as a lock.
- flags
    - PVMW_SYNC
        - Ignoring strict checks.
    - PVMW_MIGRATION
        - Find the migration entry.

consultation

- Rmap -1- (Reverse Mapping) (http://jake.dothome.co.kr/rmap-1) | 문c
- Rmap -2- (TTU & Rmap Walk) (http://jake.dothome.co.kr/rmap-2) | 문c
- Rmap -3- (PVMW) (http://jake.dothome.co.kr/rmap-3) | Sentence C – Current post

email *

Website

WRITE A COMMENT

Debug Memory -5- (Fault Injection) (http://jake.dothome.co.kr/debug-mem-5/)

Rmap -2- (TTU & Rmap Walk) (http://jake.dothome.co.kr/rmap-2/)