Rmap -2- (TTU & Rmap Walk)

2019-09-10 (http://jake.dothome.co.kr/rmap-2/) Moon Young-il (http://jake.dothome.co.kr/author/admin/) Linux Kernel (http://jake.dothome.co.kr/category/linux/)

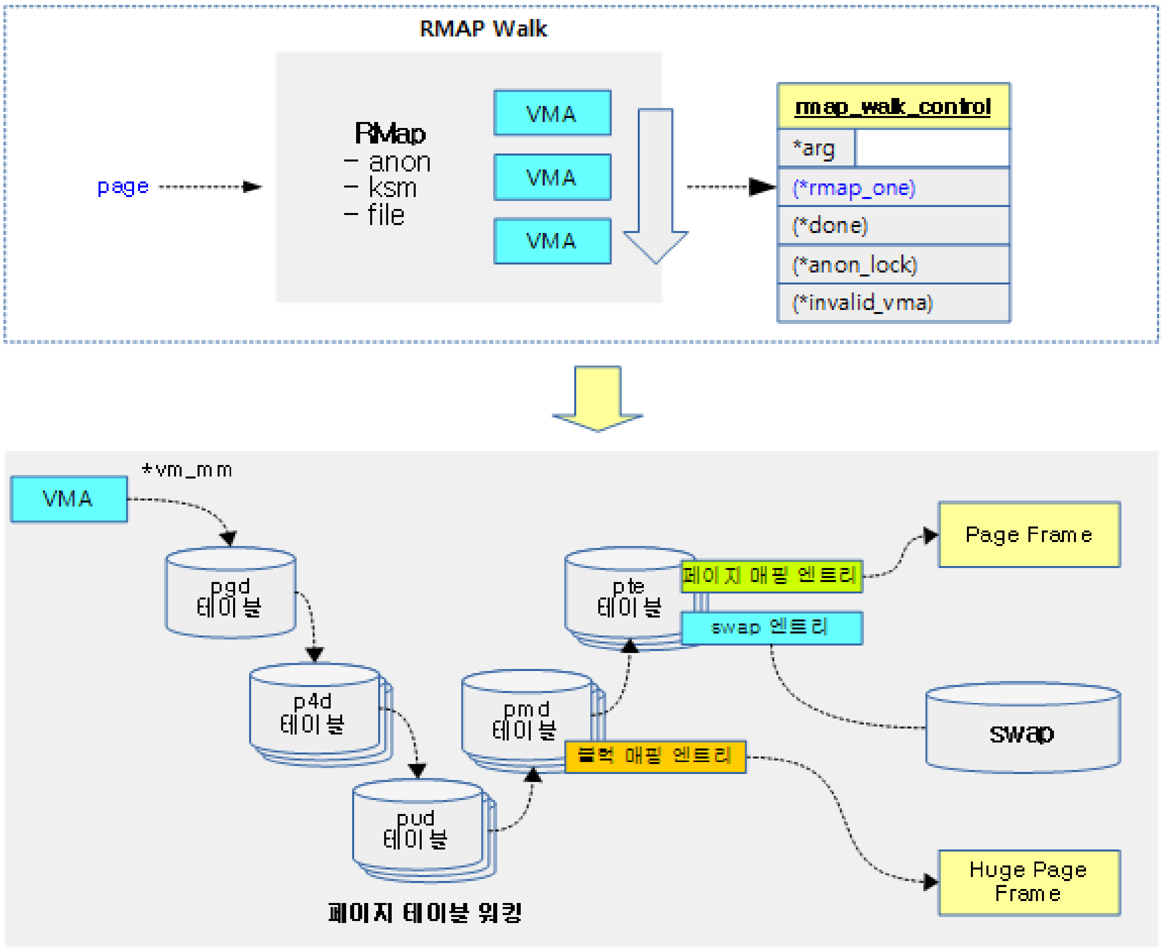
<kernel v5.0>

Rmap -2- (TTU & Rmap Walk)

Rmap Walk

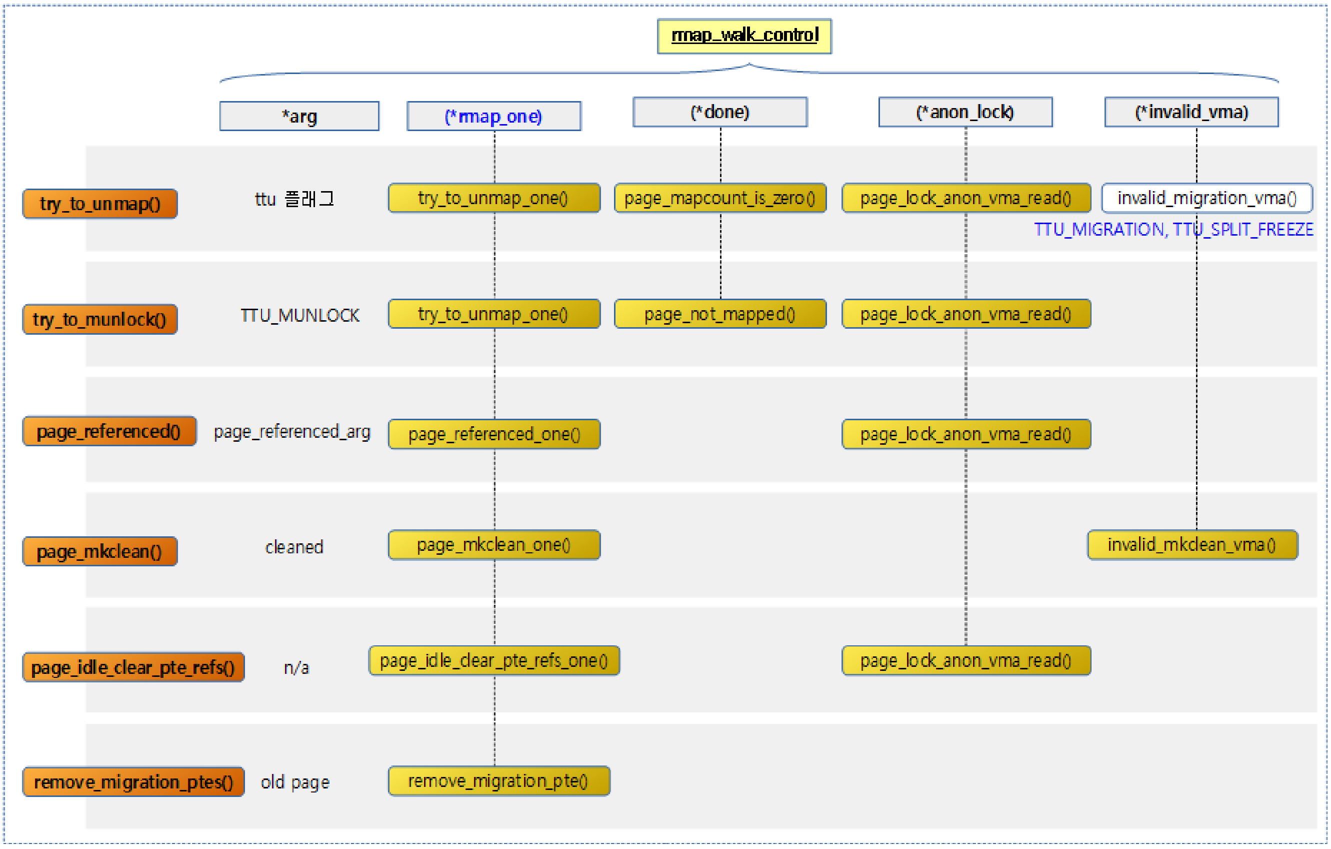
The following illustration shows the process of performing an rmap walk on a single user page and calling the hook function contained in the rmap_walk_control structure.

- If a single user page is mapped to multiple virtual addresses, it can be used to find related VMAs and target the mapped pte entries for unmapping, migrating, migrating, ... and many other functions.



(http://jake.dothome.co.kr/wp-content/uploads/2019/09/rmap-13.png)

The following figure shows the calling functions that currently use rmap walk, and shows the hook functions used.



(http://jake.dothome.co.kr/wp-content/uploads/2019/09/rmap-14a.png)

TTU(Try To Unmap)

When unmapping a user page, it is unmapped via rmap walk.

- When unmapping user space, the interlocked secondary MMU is controlled via the mmu notifier.

TTU Flag

These are the flags to use when the TTU is activated.

- TTU_MIGRATION
  - Migration Mode
- TTU_MUNLOCK
  - VMas that are not VM_LOCKED in munlock mode are skipped.
- TTU_SPLIT_HUGE_PMD
  - If the page is a huge PMD, split it.
- TTU_IGNORE_MLOCK
  - Ignoring MLOCK
- TTU_IGNORE_ACCESS
  - If the young page has ever been accessed, it will clear the access flag of the pte entry and flush the TLB, ignoring the objection.
- TTU_IGNORE_HWPOISON
  - Ignore hwpoison and use corrupted pages.
- TTU_BATCH_FLUSH

    - If possible, TLB flushes are processed at the end.
- TTU_RMAP_LOCKED
- TTU_SPLIT_FREEZE
    - When you split THP, you freeze PTE.

try_to_unmap()

mm/rmap.c

```
/**
 * try_to_unmap - try to remove all page table mappings to a page
 * @page: the page to get unmapped
 * @flags: action and flags
 *
 * Tries to remove all the page table entries which are mapping this
 * page, used in the pageout path.  Caller must hold the page lock.
 *
 * If unmap is successful, return true. Otherwise, false.
 */
```

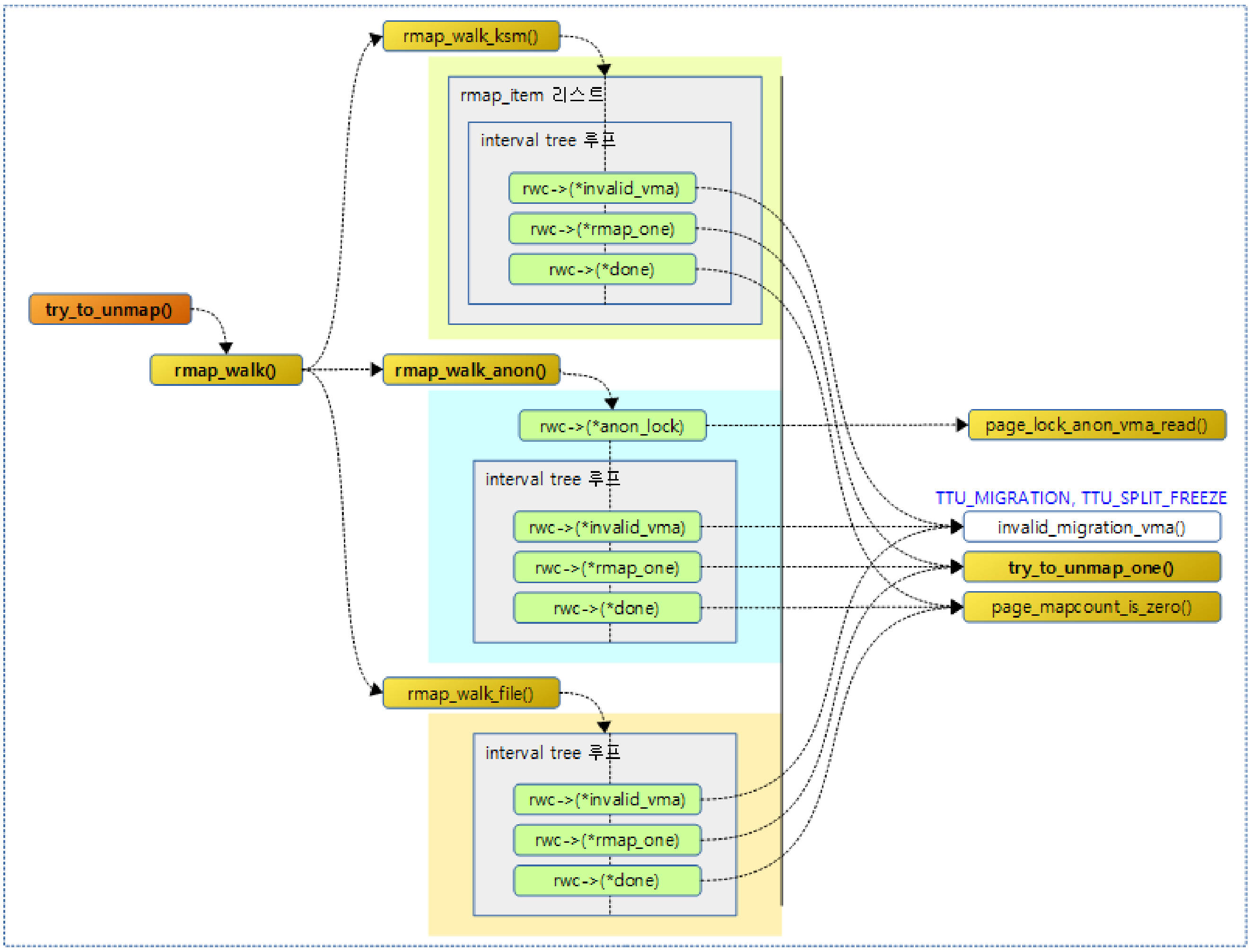
```
01 bool try_to_unmap(struct page *page, enum ttu_flags flags)
02 {
03         struct rmap_walk_control rwc = {
04                 .rmap_one = try_to_unmap_one,
05                 .arg = (void *)flags,
06                 .done = page_mapcount_is_zero,
07                 .anon_lock = page_lock_anon_vma_read,
08         };
09 
10         /*
11          * During exec, a temporary VMA is setup and later moved.
12          * The VMA is moved under the anon_vma lock but not the
13          * page tables leading to a race where migration cannot
14          * find the migration ptes. Rather than increasing the
15          * locking requirements of exec(), migration skips
16          * temporary VMAs until after exec() completes.
17          */
18         if ((flags & (TTU_MIGRATION|TTU_SPLIT_FREEZE))
19             && !PageKsm(page) && PageAnon(page))
20                 rwc.invalid_vma = invalid_migration_vma;
21 
22         if (flags & TTU_RMAP_LOCKED)
23                 rmap_walk_locked(page, &rwc);
24         else
25                 rmap_walk(page, &rwc);
26 
27         return !page_mapcount(page) ? true : false;
28 }
```

Unmap all mapping to the page via the rmap walk.

- In lines 3~8 of code, specify the hook functions to be called via RWC.
- It is used in Split or Migration (TTU_MIGRATION) of THP(TTU_SPLIT_FREEZE) in line 18~20 of the code, and for ANON mapping pages except KSM, specify the invalid_migration_vma() function to the (*invalid_vma) hook function.
    - 참고: mm: thp: introduce separate TTU flag for thp freezing (https://github.com/torvalds/linux/commit/b5ff8161e37cef3265e186ecded23324e4dc2973)
- In lines 22~25 of the code, perform the rmap walk to unmap all the pages using rwc. If the TTU_RMAP_LOCKED flag is used, the lock has been obtained from the outside.

- Note: rmap: introduce rmap_walk_locked() (https://github.com/torvalds/linux/commit/b97731992d00f09456726bfc5ab6641c07773038#diff-82d4b79a51478b4ef923b8d2f2ffdee6)

The following figure shows the process of the try_to_unmap() function.



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

page_mapcount_is_zero()

mm/rmap.c

```
static int page_mapcount_is_zero(struct page *page)
{
        return !total_mapcount(page);
}
```

Returns whether the mapping counter on the page is zero.

- When used in conjunction with the (*done) hook function of the rmap_walk_control struct, it is used to complete the rmap walk when the unmapping of the page is complete.

total_mapcount()

mm/huge_memory.c

```
int total_mapcount(struct page *page)
{
        int i, compound, ret;

        VM_BUG_ON_PAGE(PageTail(page), page);

        if (likely(!PageCompound(page)))
                return atomic_read(&page->_mapcount) + 1;

        compound = compound_mapcount(page);
        if (PageHuge(page))
                return compound;
        ret = compound;
        for (i = 0; i < HPAGE_PMD_NR; i++)
                ret += atomic_read(&page[i]._mapcount) + 1;
        /* File pages has compound_mapcount included in _mapcount */
        if (!PageAnon(page))
                return ret - compound * HPAGE_PMD_NR;
        if (PageDoubleMap(page))
                ret -= HPAGE_PMD_NR;
        return ret;
}
```

Returns the value of the mapping counter on the page.

page_lock_anon_vma_read()

mm/rmap.c

```
/*
 * Similar to page_get_anon_vma() except it locks the anon_vma.
 *
 * Its a little more complex as it tries to keep the fast path to a single
 * atomic op -- the trylock. If we fail the trylock, we fall back to getting a
 * reference like with page_get_anon_vma() and then block on the mutex.
 */
```

```
struct anon_vma *page_lock_anon_vma_read(struct page *page)
{
        struct anon_vma *anon_vma = NULL;
        struct anon_vma *root_anon_vma;
        unsigned long anon_mapping;

        rcu_read_lock();
        anon_mapping = (unsigned long)READ_ONCE(page->mapping);
        if ((anon_mapping & PAGE_MAPPING_FLAGS) != PAGE_MAPPING_ANON)
                goto out;
        if (!page_mapped(page))
                goto out;

        anon_vma = (struct anon_vma *) (anon_mapping - PAGE_MAPPING_ANON);
        root_anon_vma = READ_ONCE(anon_vma->root);
        if (down_read_trylock(&root_anon_vma->rwsem)) {
                /*
                 * If the page is still mapped, then this anon_vma is still
                 * its anon_vma, and holding the mutex ensures that it will
                 * not go away, see anon_vma_free().
                 */
                if (!page_mapped(page)) {
                        up_read(&root_anon_vma->rwsem);
                        anon_vma = NULL;
                }
                goto out;
        }

        /* trylock failed, we got to sleep */
        if (!atomic_inc_not_zero(&anon_vma->refcount)) {
                anon_vma = NULL;
                goto out;
        }

        if (!page_mapped(page)) {
                rcu_read_unlock();
                put_anon_vma(anon_vma);
                return NULL;
        }

        /* we pinned the anon_vma, its safe to sleep */
        rcu_read_unlock();
        anon_vma_lock_read(anon_vma);

        if (atomic_dec_and_test(&anon_vma->refcount)) {
                /*
                 * Oops, we held the last refcount, release the lock
                 * and bail -- can't simply use put_anon_vma() because
                 * we'll deadlock on the anon_vma_lock_write() recursion.
                 */
                anon_vma_unlock_read(anon_vma);
                __put_anon_vma(anon_vma);
                anon_vma = NULL;
        }

        return anon_vma;

out:
        rcu_read_unlock();
        return anon_vma;
}
```

Gets a root anon_vma lock for the anon page, and returns a anon_vma.

rmap Walk

rmap_walk()

mm/rmap.c

```
void rmap_walk(struct page *page, struct rmap_walk_control *rwc)
{
        if (unlikely(PageKsm(page)))
                rmap_walk_ksm(page, rwc);
        else if (PageAnon(page))
                rmap_walk_anon(page, rwc, false);
        else
                rmap_walk_file(page, rwc, false);
}
```

Traversing the VMAs for the KSM, ANON, and FILE types to which the page belongs, and executing RWC's (*rmap_one) hook function.

```
01 /*
02  * rmap_walk_anon - do something to anonymous page using the object-based
03  * rmap method
04  * @page: the page to be handled
05  * @rwc: control variable according to each walk type
06  *
07  * Find all the mappings of a page using the mapping pointer and the vma chains
08  * contained in the anon_vma struct it points to.
09  *
10  * When called from try_to_munlock(), the mmap_sem of the mm containing the vma
11  * where the page was found will be held for write.  So, we won't recheck
12  * vm_flags for that VMA.  That should be OK, because that vma shouldn't be
13  * LOCKED.
14  */
```

```
01 static void rmap_walk_anon(struct page *page, struct rmap_walk_control *rwc,
02                 bool locked)
03 {
04         struct anon_vma *anon_vma;
05         pgoff_t pgoff_start, pgoff_end;
06         struct anon_vma_chain *avc;
07 
08         if (locked) {
09                 anon_vma = page_anon_vma(page);
10                 /* anon_vma disappear under us? */
11                 VM_BUG_ON_PAGE(!anon_vma, page);
12         } else {
13                 anon_vma = rmap_walk_anon_lock(page, rwc);
14         }
15         if (!anon_vma)
16                 return;
17 
18         pgoff_start = page_to_pgoff(page);
19         pgoff_end = pgoff_start + hpage_nr_pages(page) - 1;
20         anon_vma_interval_tree_foreach(avc, &anon_vma->rb_root,
21                         pgoff_start, pgoff_end) {
22                 struct vm_area_struct *vma = avc->vma;
23                 unsigned long address = vma_address(page, vma);
24 
25                 cond_resched();
26 
27                 if (rwc->invalid_vma && rwc->invalid_vma(vma, rwc->arg))
28                         continue;
29 
30                 if (!rwc->rmap_one(page, vma, address, rwc->arg))
31                         break;
32                 if (rwc->done && rwc->done(page))
33                         break;
34         }
35 
36         if (!locked)
37                 anon_vma_unlock_read(anon_vma);
38 }
```

It traverses the Anon VMAs to which Page belongs, executing RWC's (*rmap_one) hook function.

- Get the anon_vma corresponding to the page from lines 8~16 of the code. If no @locked is set, this function must obtain a lock on the anon_vma.
- Start pgoff at code lines 18~23 ~ traverse the VMAs belonging to the anon_vma included at the end of pgoff.

- If the result of performing the (*invalid_vma) hook function on RWC in lines 27~28 is ture, the VMA should be skipped.
- On lines 30~31 of the code, perform RWC's (*rmap_one) hook function. If it fails, break it.
- After performing RWC's (*done) hook function on lines 32~33 of code, break if the result is true.
- Unlock the anon_vma in code lines 36~37.

```
01 /*
02  * rmap_walk_file - do something to file page using the object-based rma
   p method
03  * @page: the page to be handled
04  * @rwc: control variable according to each walk type
05  *
06  * Find all the mappings of a page using the mapping pointer and the vma
   chains
07  * contained in the address_space struct it points to.
08  *
09  * When called from try_to_munlock(), the mmap_sem of the mm containing
   the vma
10  * where the page was found will be held for write.  So, we won't rechec
   k
11  * vm_flags for that VMA.  That should be OK, because that vma shouldn't
   be
12  * LOCKED.
13  */
```

```
01 static void rmap_walk_file(struct page *page, struct rmap_walk_control *
   rwc,
02                 bool locked)
03 {
04         struct address_space *mapping = page_mapping(page);
05         pgoff_t pgoff_start, pgoff_end;
06         struct vm_area_struct *vma;
07 
08         /*
09          * The page lock not only makes sure that page->mapping cannot
10          * suddenly be NULLified by truncation, it makes sure that the
11          * structure at mapping cannot be freed and reused yet,
12          * so we can safely take mapping->i_mmap_rwsem.
13          */
14         VM_BUG_ON_PAGE(!PageLocked(page), page);
15 
16         if (!mapping)
17                 return;
18 
19         pgoff_start = page_to_pgoff(page);
20         pgoff_end = pgoff_start + hpage_nr_pages(page) - 1;
21         if (!locked)
22                 i_mmap_lock_read(mapping);
23         vma_interval_tree_foreach(vma, &mapping->i_mmap,
24                         pgoff_start, pgoff_end) {
25                 unsigned long address = vma_address(page, vma);
26 
27                 cond_resched();
28 
29                 if (rwc->invalid_vma && rwc->invalid_vma(vma, rwc->arg))
30                         continue;
31 
32                 if (!rwc->rmap_one(page, vma, address, rwc->arg))
33                         goto done;
34                 if (rwc->done && rwc->done(page))
35                         goto done;
36         }
37 
38 done:
39         if (!locked)
40                 i_mmap_unlock_read(mapping);
41 }
```

It traverses the file VMAs to which the page belongs, and executes RWC's (*rmap_one) hook function.

- If it is not a mapped file page in line 4~17 of the code, exit the function.
- In line 19~20 of the code, the file page calculates the pgoff start and the pgoff end.
- If the @locked is not set in line 21~22 of the code, the lock on the i_mmap must be obtained from this function.
- Start pgoff at code lines 23~24 ~ traverse the VMAs through the interval tree for the file mapping space included at the end of pgoff.
- If the result of RWC's (*invalid_vma) hook function on lines 29~30 is true, the VMA is skipped.
- In lines 32~33 of code, perform RWC's (*rmap_one) hook function. If the result of the performance fails, break it.
- If the result of RWC's (*done) hook function in line 34~35 is true, break it.
- In code lines 38~40, the done: label is. Unlock the i_mmap.

Try 1 unmap

try_to_unmap_one()

mm/rmap.c -1/6-

```
/*
 * @arg: enum ttu_flags will be passed to this argument
 */
```

```
static bool try_to_unmap_one(struct page *page, struct vm_area_struct *vma,
                     unsigned long address, void *arg)
{
        struct mm_struct *mm = vma->vm_mm;
        struct page_vma_mapped_walk pvmw = {
                .page = page,
                .vma = vma,
                .address = address,
        };
        pte_t pteval;
        struct page *subpage;
        bool ret = true;
        struct mmu_notifier_range range;
        enum ttu_flags flags = (enum ttu_flags)arg;

        /* munlock has nothing to gain from examining un-locked vmas */
        if ((flags & TTU_MUNLOCK) && !(vma->vm_flags & VM_LOCKED))
                return true;

        if (IS_ENABLED(CONFIG_MIGRATION) && (flags & TTU_MIGRATION) &&
            is_zone_device_page(page) && !is_device_private_page(page))
                return true;

        if (flags & TTU_SPLIT_HUGE_PMD) {
                split_huge_pmd_address(vma, address,
                                flags & TTU_SPLIT_FREEZE, page);
        }

        /*
```

```
30           * For THP, we have to assume the worse case ie pmd for invalidation.
31           * For hugetlb, it could be much worse if we need to do pud
32           * invalidation in the case of pmd sharing.
33           *
34           * Note that the page can not be free in this function as call of
35           * try_to_unmap() must hold a reference on the page.
36           */
37          mmu_notifier_range_init(&range, vma->vm_mm, address,
38                                  min(vma->vm_end, address +
39                                      (PAGE_SIZE << compound_order(page))));
40          if (PageHuge(page)) {
41                  /*
42                   * If sharing is possible, start and end will be adjusted
43                   * accordingly.
44                   */
45                  adjust_range_if_pmd_sharing_possible(vma, &range.start,
46                                                       &range.end);
47          }
48          mmu_notifier_invalidate_range_start(&range);
```

- In line 4 of the code, we get the virtual address space management mm to which the VMA belongs.
- Prepare a pvmw to check whether it is mapped in code lines 5~9.
- If there is a TTU_MUNLOCK request to a VMA that is not VM_LOCKED set in lines 17~18 of code, just to skip that page and return success.
- In line 20~22 of the code, when requesting the migration mapping, it returns true to skip if it is a zone device and not an HMM.
  - Non-HMM Zone devices are not capable of performing normal mapping/unmapping operations.
- On lines 24~27 of the code, perform a huge page split request.
- Initialize the mmu_notifier_range in lines 37~39 of code.
- In line 40~47 of the code, adjust the range above if it is a huge page.
- In line 48 of the code, the MMU Notifier calls the registered (*invalidate_range_start) function to let it know that it starts before performing a TLB invalidation on the Secondary MMU's range.

mm/rmap.c -2/6-

```
01          while (page_vma_mapped_walk(&pvmw)) {
02  #ifdef CONFIG_ARCH_ENABLE_THP_MIGRATION
03                  /* PMD-mapped THP migration entry */
04                  if (!pvmw.pte && (flags & TTU_MIGRATION)) {
05                          VM_BUG_ON_PAGE(PageHuge(page) || !PageTransCompound(page), page);
06
07                          set_pmd_migration_entry(&pvmw, page);
08                          continue;
09                  }
10  #endif
11
12                  /*
13                   * If the page is mlock()d, we cannot swap it out.
14                   * If it's recently referenced (perhaps page_referenced
		 * skipped over this mm) then we should reactivate it.
		 */
		if (!(flags & TTU_IGNORE_MLOCK)) {
			if (vma->vm_flags & VM_LOCKED) {
				/* PTE-mapped THP are never mlocked */
				if (!PageTransCompound(page)) {
					/*
					 * Holding pte lock, we do *not* need
					 * mmap_sem here
					 */
					mlock_vma_page(page);
				}
				ret = false;
				page_vma_mapped_walk_done(&pvmw);
				break;
			}
			if (flags & TTU_MUNLOCK)
				continue;
		}

		/* Unexpected PMD-mapped THP? */
		VM_BUG_ON_PAGE(!pvmw.pte, page);

		subpage = page - page_to_pfn(page) + pte_pfn(*pvmw.pte);
		address = pvmw.address;

		if (PageHuge(page)) {
			if (huge_pmd_unshare(mm, &address, pvmw.pte)) {
				/*
				 * huge_pmd_unshare unmapped an entire PMD
				 * page.  There is no way of knowing exactly
				 * which PMDs may be cached for this mm, so
				 * we must flush them all.  start/end were
				 * already adjusted above to cover this range.
				 */
				flush_cache_range(vma, range.start, range.end);
				flush_tlb_range(vma, range.start, range.end);
				mmu_notifier_invalidate_range(mm, range.start,
							      range.end);

				/*
				 * The ref count of the PMD page was dropped
				 * which is part of the way map counting
				 * is done for shared PMDs.  Return 'true'
				 * here.  When there is no other sharing,
				 * huge_pmd_unshare returns false and we will
				 * unmap the actual page and drop map count
				 * to zero.
				 */
				page_vma_mapped_walk_done(&pvmw);
				break;
			}
		}
```

- It loops only if the normal mapping requested via pvmw in line 1 is healthy.
- In line 4~9 of the code, map the thp page to the migration entry via the pmd entry and continue the loop.
- If a request is made without the TTU_IGNORE_MLOCK flag in lines 17~33 of code, the mlocked page cannot be swapped out. VM_LOCKED If it's a VMA area, stop the loop and stop processing.
- In line 38~67 of code, if it is a huge page that is not shared, flush the cache and tlb cache for the range area, and perform tlb invalidation for the range of the secondary MMU. Then stop the loop and stop processing.

mm/rmap.c -3/6-

```
01 .               if (IS_ENABLED(CONFIG_MIGRATION) &&
02                     (flags & TTU_MIGRATION) &&
03                     is_zone_device_page(page)) {
04                         swp_entry_t entry;
05                         pte_t swp_pte;
06 
07                         pteval = ptep_get_and_clear(mm, pvmw.address, pvmw.pte);
08 
09                         /*
10                          * Store the pfn of the page in a special migration
11                          * pte. do_swap_page() will wait until the migration
12                          * pte is removed and then restart fault handling.
13                          */
14                         entry = make_migration_entry(page, 0);
15                         swp_pte = swp_entry_to_pte(entry);
16                         if (pte_soft_dirty(pteval))
17                                 swp_pte = pte_swp_mksoft_dirty(swp_pte);
18                         set_pte_at(mm, pvmw.address, pvmw.pte, swp_pte);
19                         /*
20                          * No need to invalidate here it will synchronize on
21                          * against the special swap migration pte.
22                          */
23                         goto discard;
24                 }
25 
26                 if (!(flags & TTU_IGNORE_ACCESS)) {
27                         if (ptep_clear_flush_young_notify(vma, address,
28                                                 pvmw.pte)) {
29                                 ret = false;
30                                 page_vma_mapped_walk_done(&pvmw);
31                                 break;
32                         }
33                 }
34 
35                 /* Nuke the page table entry. */
36                 flush_cache_page(vma, address, pte_pfn(*pvmw.pte));
37                 if (should_defer_flush(mm, flags)) {
38                         /*
39                          * We clear the PTE but do not flush so potentially
40                          * a remote CPU could still be writing to the page.
41                          * If the entry was previously clean then the
42                          * architecture must guarantee that a clear->dirty
43                              * transition on a cached TLB entry is written through
44                              * and traps if the PTE is unmapped.
45                              */
46                             pteval = ptep_get_and_clear(mm, address, pvmw.pte);
47 
48                             set_tlb_ubc_flush_pending(mm, pte_dirty(pteval));
49                     } else {
50                             pteval = ptep_clear_flush(vma, address, pvmw.pte);
51                     }
52 
53                     /* Move the dirty bit to the page. Now the pte is gone. */
54                     if (pte_dirty(pteval))
55                             set_page_dirty(page);
56 
57                     /* Update high watermark before we lower rss */
58                     update_hiwater_rss(mm);
```

- In line 1~24 of the code, the TTU_MIGRATION flag is requested on the zone device page. Map the migration information to the PTE entry by creating a swap entry.
  - swap entries are used to map and use migration information. This mapping allows the fault handler to wait for the swap fault do_swap_page() to complete the migration.
  - The Soft Dirty feature is currently only available on the x86_64, powerpc_64, and S390 architectures.
  - 참고: mm/migrate: support un-addressable ZONE_DEVICE page in migration (https://github.com/torvalds/linux/commit/a5430dda8a3a1cdd532e37270e6f36436241b6e7)
- If there is no TTU_IGNORE_ACCESS flag request in line 26~33 of the code, test-and-clear the young/accessed flag of the secondary MMU's pte entry corresponding to the address, and then flush it if it has been accessed and let the routine abort.
- In line 36 of code, flush the cache for the user virtual address.
  - The ARM64 architecture does nothing.
  - If the cache type of the architecture is used for vivt or vipt aliasing, it should be flush.
- Clear and unmap the PTE entry mapped to the user's virtual address in code lines 37~51 and flush it with TLB. If a TTU_BATCH_FLUSH plug request is received, the TLB flush will be collected and processed at the end to improve performance.
- If the existing PTE entry is in a dirty state before unmapping (clearing) in code lines 54~55, set the page to a dirty state.
- Update the hiwater_rss counter in mm at code line 58 if it is the highest value.
  - Number of mm file pages + number of anon pages + number of shmem pages

mm/rmap.c -4/6-

```
01 .                   if (PageHWPoison(page) && !(flags & TTU_IGNORE_HWPOISON)) {
02                             pteval = swp_entry_to_pte(make_hwpoison_entry(subpage));
03                             if (PageHuge(page)) {
04                                     int nr = 1 << compound_order(page);
05                                     hugetlb_count_sub(nr, mm);
```

```
                                set_huge_swap_pte_at(mm, address,
                                                     pvmw.pte, pteval,
                                                     vma_mmu_pagesize(vma));
                        } else {
                                dec_mm_counter(mm, mm_counter(page));
                                set_pte_at(mm, address, pvmw.pte, pteval);
                        }

                } else if (pte_unused(pteval) && !userfaultfd_armed(vma)) {
                        /*
                         * The guest indicated that the page content is of no
                         * interest anymore. Simply discard the pte, vmscan
                         * will take care of the rest.
                         * A future reference will then fault in a new zero
                         * page. When userfaultfd is active, we must not drop
                         * this page though, as its main user (postcopy
                         * migration) will not expect userfaults on already
                         * copied pages.
                         */
                        dec_mm_counter(mm, mm_counter(page));
                        /* We have to invalidate as we cleared the pte */
                        mmu_notifier_invalidate_range(mm, address,
                                                      address + PAGE_SIZE);
                } else if (IS_ENABLED(CONFIG_MIGRATION) &&
                                (flags & (TTU_MIGRATION|TTU_SPLIT_FREEZE))) {
                        swp_entry_t entry;
                        pte_t swp_pte;

                        if (arch_unmap_one(mm, vma, address, pteval) < 0) {
                                set_pte_at(mm, address, pvmw.pte, pteval);
                                ret = false;
                                page_vma_mapped_walk_done(&pvmw);
                                break;
                        }

                        /*
                         * Store the pfn of the page in a special migration
                         * pte. do_swap_page() will wait until the migration
                         * pte is removed and then restart fault handling.
                         */
                        entry = make_migration_entry(subpage,
                                        pte_write(pteval));
                        swp_pte = swp_entry_to_pte(entry);
                        if (pte_soft_dirty(pteval))
                                swp_pte = pte_swp_mksoft_dirty(swp_pte);
                        set_pte_at(mm, address, pvmw.pte, swp_pte);
                        /*
                         * No need to invalidate here it will synchronize on
                         * against the special swap migration pte.
                         */
```

- In line 1~12 of the code, this is the hwpoison page and the request does not use the TTU_IGNORE_HWPOISON flag. Reduce the relevant MM counters (anon, file, shm) by the number of pages. Then map the swap entry value to the PTE entry.
- In code lines 14~28, if the userfaultfd vma is a deprecated PTE value, the relevant MM counters (anon, file, shm) are decremented by the number of pages. The secondary MMU is then also required to perform TLB invalidates on the virtual address.
- This is the case when a TTU_MIGRATION or TTU_SPLIT_FREEZE flag request is received on code lines 29~51. Map the swap entry to be migrated. If soft dirty is set in an existing mapping, it will also be included in the swap entry.

mm/rmap.c -5/6-

```
01 .               } else if (PageAnon(page)) {
02                         swp_entry_t entry = { .val = page_private(subpage) };
03                         pte_t swp_pte;
04                         /*
05                          * Store the swap location in the pte.
06                          * See handle_pte_fault() ...
07                          */
08                         if (unlikely(PageSwapBacked(page) != PageSwapCache(page))) {
09                                 WARN_ON_ONCE(1);
10                                 ret = false;
11                                 /* We have to invalidate as we cleared the pte */
12                                 mmu_notifier_invalidate_range(mm, address,
13                                                         address + PAGE_SIZE);
14                                 page_vma_mapped_walk_done(&pvmw);
15                                 break;
16                         }
17 
18                         /* MADV_FREE page check */
19                         if (!PageSwapBacked(page)) {
20                                 if (!PageDirty(page)) {
21                                         /* Invalidate as we cleared the pte */
22                                         mmu_notifier_invalidate_range(mm,
23                                                 address, address + PAGE_SIZE);
24                                         dec_mm_counter(mm, MM_ANONPAGES);
25                                         goto discard;
26                                 }
27 
28                                 /*
29                                  * If the page was redirtied, it cannot be
30                                  * discarded. Remap the page to page table.
31                                  */
32                                 set_pte_at(mm, address, pvmw.pte, pteval);
33                                 SetPageSwapBacked(page);
34                                 ret = false;
35                                 page_vma_mapped_walk_done(&pvmw);
36                                 break;
37                         }
```

```
38
39                          if (swap_duplicate(entry) < 0) {
40                                  set_pte_at(mm, address, pvmw.pte, pteva
l);
41                                  ret = false;
42                                  page_vma_mapped_walk_done(&pvmw);
43                                  break;
44                          }
45                          if (arch_unmap_one(mm, vma, address, pteval) <
0) {
46                                  set_pte_at(mm, address, pvmw.pte, pteva
l);
47                                  ret = false;
48                                  page_vma_mapped_walk_done(&pvmw);
49                                  break;
50                          }
51                          if (list_empty(&mm->mmlist)) {
52                                  spin_lock(&mmlist_lock);
53                                  if (list_empty(&mm->mmlist))
54                                          list_add(&mm->mmlist, &init_mm.m
mlist);
55                                  spin_unlock(&mmlist_lock);
56                          }
57                          dec_mm_counter(mm, MM_ANONPAGES);
58                          inc_mm_counter(mm, MM_SWAPENTS);
59                          swp_pte = swp_entry_to_pte(entry);
60                          if (pte_soft_dirty(pteval))
61                                  swp_pte = pte_swp_mksoft_dirty(swp_pte);
62                          set_pte_at(mm, address, pvmw.pte, swp_pte);
63                          /* Invalidate as we cleared the pte */
64                          mmu_notifier_invalidate_range(mm, address,
65                                                        address + PAGE_SIZ
E);
```

- The code is processed on lines 1~16 for the anon page. In a small probability, if the page does not match the swapbacked and swapcache flag settings, it will perform TLB invalidation of the secondary MMU and abort the routine.
- In line 19~37 of code, if the page is not swapbacked, map the page again and abort the routine. However, if it's not a dirty page, perform TLB invalidation of the secondary MMU, decrement the Anon MM counter, and then move to the Discard label to proceed with the following:
- In lines 39~44 of code, increment the reference counter of the swap entry by 1. If it's an error, it remaps the page and aborts the routine.
- If there is an error in the architecture-specific unmap in line 45~50, it will remap the page and break the routine.
  - Currently, only sparc_64 architectures are supported.
- If the current mmlist is empty in code lines 51~56, add it to the mmlist in init_mm.
- In code lines 57~65, increment the ANON and Swap MM counters, map the swap entry, and perform TLB invalidation of the secondary MMU.

mm/rmap.c -6/6-

```
01                  } else {
02                          /*
03                           * This is a locked file-backed page, thus it ca
nnot
04                           * be removed from the page cache and replaced b
y a new
```

```
05                               * page before mmu_notifier_invalidate_range_end, so no
06                               * concurrent thread might update its page table to
07                               * point at new page while a device still is using this
08                               * page.
09                               *
10                               * See Documentation/vm/mmu_notifier.rst
11                               */
12                              dec_mm_counter(mm, mm_counter_file(page));
13                      }
14 discard:
15                      /*
16                       * No need to call mmu_notifier_invalidate_range() it has be
17                       * done above for all cases requiring it to happen under page
18                       * table lock before mmu_notifier_invalidate_range_end()
19                       *
20                       * See Documentation/vm/mmu_notifier.rst
21                       */
22                      page_remove_rmap(subpage, PageHuge(page));
23                      put_page(page);
24              }
25
26              mmu_notifier_invalidate_range_end(&range);
27
28              return ret;
29 }
```

- In code lines 1~13, for other (file-backed pages), the file-related mm counters (file, shm) are decremented.
- On lines 14~23 of the code, the discard: label is. After removing the rmap mapping of the page, you are done using the page.
- In line 26 of code, the invalidate of the range area of the primary MMU was completed. So we call this function to call the (*invalidate_range_end) hook function registered in the mmu notifier for the secondary MMU.
    - (*invalidate_range_start) always works in pairs with the hook function.

ptep_clear_flush_young_notify()

include/linux/mmu_notifier.h

```
#define ptep_clear_flush_young_notify(__vma, __address, __ptep)         \
({                                                                      \
        int __young;                                                    \
        struct vm_area_struct *___vma = __vma;                          \
        unsigned long ___address = __address;                           \
        __young = ptep_clear_flush_young(___vma, ___address, __ptep);   \
        __young |= mmu_notifier_clear_flush_young(___vma->vm_mm,        \
                                                  ___address,           \
                                                  ___address +          \
10                                                        PAGE_SIZE);
   \
11          __young;
   \
12 })
```

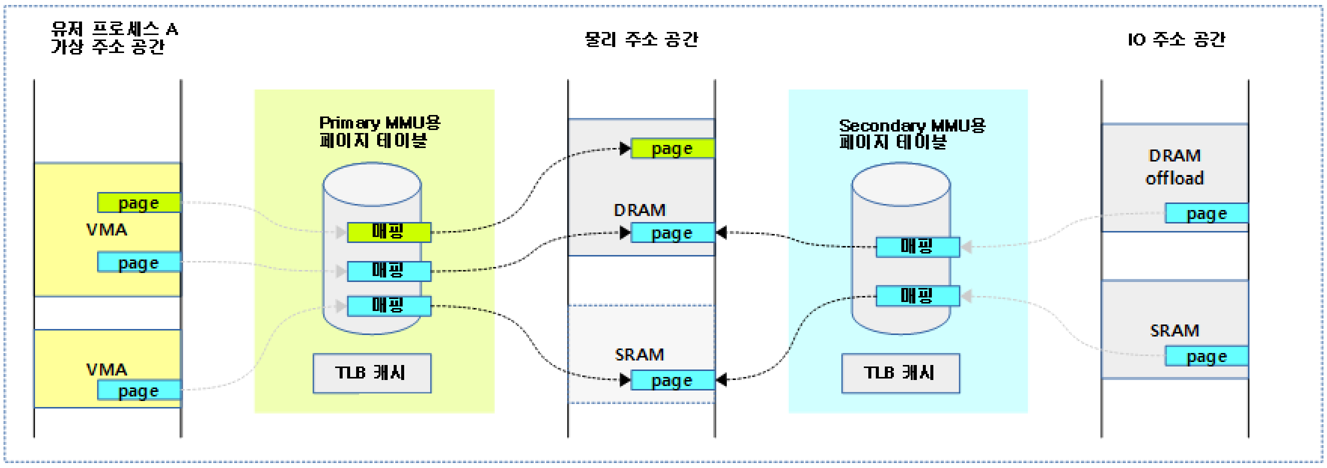
If the page has been accessed, clear the access flag and perform a flush. Returns the result of a (*clear_flush_young) hook function that has access or is registered in the mnu_notifier.

- In line 6 of the code, check whether the PTE entry has an access flag set or not, and clear it. If it has ever been accessed, it flushes the TLB.
- Add the result of calling the (*clear_flush_young) hook function registered in the MNU Notifier on line 7.
  - It is used in the following functions:
    - virt/kvm/kvm_main.c – kvm_mmu_notifier_clear_flush_young()
    - drivers/iommu/amd_iommu_v2.c – mn_clear_flush_young()

MMU Notifier

Secondary MMUs associated with physical addresses mapped to virtual address zones (kvm, iommu, ...)MMU-related operation function. Drivers who will use MMU Notifier prepare a mmu_notifier structure including a mmu_notifier_ops and register it with mmu_notifer_register().

The following figure shows how the Secondary MMU is affected when the mapping of a page in the physical address space mapped to the user virtual address space is changed.



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

mmu_notifier Struct

include/linux/mmu_notifier.h

```
/*
 * The notifier chains are protected by mmap_sem and/or the reverse map
 * semaphores. Notifier chains are only changed when all reverse maps an
d
 * the mmap_sem locks are taken.
 *
 * Therefore notifier chains can only be traversed when either
 *
 * 1. mmap_sem is held.
 * 2. One of the reverse map locks is held (i_mmap_rwsem or anon_vma->rw
sem).
 * 3. No other concurrent thread can access the list (release)
 */
```

```
struct mmu_notifier {
        struct hlist_node hlist;
        const struct mmu_notifier_ops *ops;
};
```

It is a structure that registers the mmu_notifier_ops prepared by the IOMMU driver by adding it to the list of mmu_notifier_mm of mm that manages the relevant virtual address space.

- hlist
  - This is the node entry used when registering in the mm_struct's member mmu_notifier_mm->list.
- *ops
  - Point to the mmu_notifier_ops structure you prepared.

mmu_notifier_ops Struct

include/linux/mmu_notifier.h -1/2-

```
struct mmu_notifier_ops {
        /*
         * Called either by mmu_notifier_unregister or when the mm is
         * being destroyed by exit_mmap, always before all pages are
         * freed. This can run concurrently with other mmu notifier
         * methods (the ones invoked outside the mm context) and it
         * should tear down all secondary mmu mappings and freeze the
         * secondary mmu. If this method isn't implemented you've to
         * be sure that nothing could possibly write to the pages
         * through the secondary mmu by the time the last thread with
         * tsk->mm == mm exits.
         *
         * As side note: the pages freed after ->release returns could
         * be immediately reallocated by the gart at an alias physical
         * address with a different cache model, so if ->release isn't
         * implemented because all _software_ driven memory accesses
         * through the secondary mmu are terminated by the time the
         * last thread of this mm quits, you've also to be sure that
         * speculative _hardware_ operations can't allocate dirty
         * cachelines in the cpu that could not be snooped and made
         * coherent with the other read and write operations happening
         * through the gart alias address, so leading to memory
         * corruption.
         */
        void (*release)(struct mmu_notifier *mn,
                        struct mm_struct *mm);

        /*
         * clear_flush_young is called after the VM is
         * test-and-clearing the young/accessed bitflag in the
         * pte. This way the VM will provide proper aging to the
         * accesses to the page through the secondary MMUs and not
         * only to the ones through the Linux pte.
         * Start-end is necessary in case the secondary MMU is mapping t
he page
         * at a smaller granularity than the primary MMU.
         */
        int (*clear_flush_young)(struct mmu_notifier *mn,
                                 struct mm_struct *mm,
                                 unsigned long start,
                                 unsigned long end);

        /*
         * clear_young is a lightweight version of clear_flush_young. Like the
         * latter, it is supposed to test-and-clear the young/accessed bitflag
         * in the secondary pte, but it may omit flushing the secondary tlb.
         */
        int (*clear_young)(struct mmu_notifier *mn,
                           struct mm_struct *mm,
                           unsigned long start,
                           unsigned long end);

        /*
         * test_young is called to check the young/accessed bitflag in
         * the secondary pte. This is used to know if the page is
         * frequently used without actually clearing the flag or tearing
         * down the secondary mapping on the page.
         */
        int (*test_young)(struct mmu_notifier *mn,
                          struct mm_struct *mm,
                          unsigned long address);

        /*
         * change_pte is called in cases that pte mapping to page is changed:
         * for example, when ksm remaps pte to point to a new shared page.
         */
        void (*change_pte)(struct mmu_notifier *mn,
                           struct mm_struct *mm,
                           unsigned long address,
                           pte_t pte);
```

- (*release)
  - This is a hook function that is activated when mm is removed or when mmu_notifer_unregister() is called.
- (*clear_flush_young)
  - This is a hook function that is called after using Test-and-Clearing for the YOUNG/Accessed bit flag in the PTE entry.
  - Perform a test-and-clear of the young/accessed bit flags associated with the start ~ end address range on the secondary MMU and then perform a tlb flush of the secondary MMU.
- (*clear_young)
  - The light version of (*clear_flush_young) above does not perform a tlb flush of the secondary MMU.
- (*test_young)
  - Secondary MMU returns the young/accessed bit flag status associated with the start ~ end address range.
- (*change_pte)
  - Replace the pte associated with the address in the Secondary MMU.

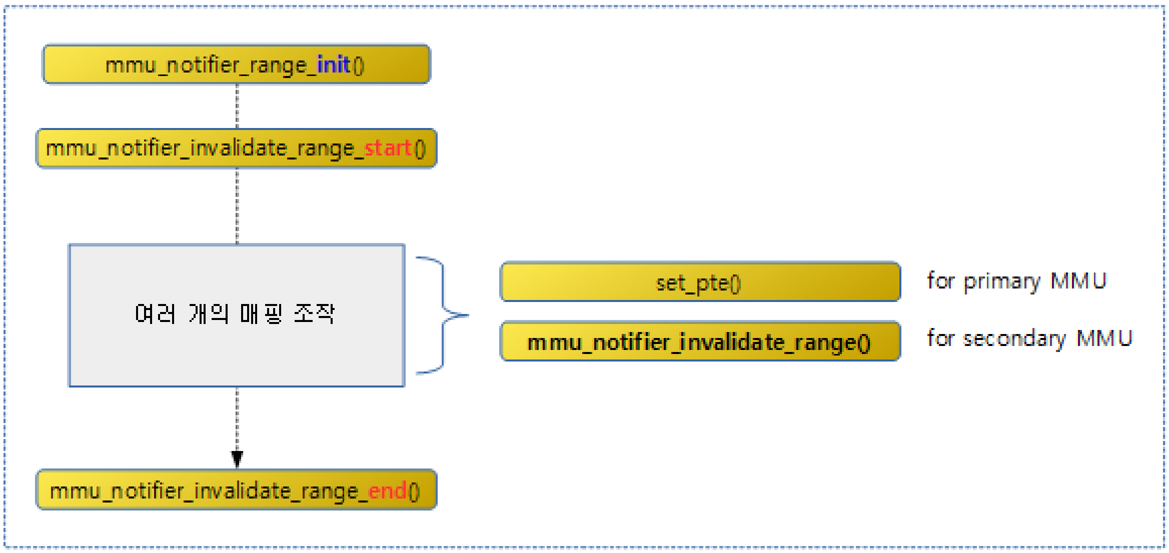
include/linux/mmu_notifier.h -2/2-

```
        /*
         * invalidate_range_start() and invalidate_range_end() must be
         * paired and are called only when the mmap_sem and/or the
         * locks protecting the reverse maps are held. If the subsystem
         * can't guarantee that no additional references are taken to
         * the pages in the range, it has to implement the
         * invalidate_range() notifier to remove any references taken
         * after invalidate_range_start().
         *
         * Invalidation of multiple concurrent ranges may be
         * optionally permitted by the driver. Either way the
         * establishment of sptes is forbidden in the range passed to
         * invalidate_range_begin/end for the whole duration of the
         * invalidate_range_begin/end critical section.
         *
         * invalidate_range_start() is called when all pages in the
         * range are still mapped and have at least a refcount of one.
         *
         * invalidate_range_end() is called when all pages in the
         * range have been unmapped and the pages have been freed by
         * the VM.
         *
         * The VM will remove the page table entries and potentially
         * the page between invalidate_range_start() and
         * invalidate_range_end(). If the page must not be freed
         * because of pending I/O or other circumstances then the
         * invalidate_range_start() callback (or the initial mapping
         * by the driver) must make sure that the refcount is kept
         * elevated.
         *
         * If the driver increases the refcount when the pages are
         * initially mapped into an address space then either
         * invalidate_range_start() or invalidate_range_end() may
         * decrease the refcount. If the refcount is decreased on
         * invalidate_range_start() then the VM can free pages as page
         * table entries are removed.  If the refcount is only
         * droppped on invalidate_range_end() then the driver itself
         * will drop the last refcount but it must take care to flush
         * any secondary tlb before doing the final free on the
         * page. Pages will no longer be referenced by the linux
         * address space but may still be referenced by sptes until
         * the last refcount is dropped.
         *
         * If blockable argument is set to false then the callback cannot
         * sleep and has to return with -EAGAIN. 0 should be returned
         * otherwise. Please note that if invalidate_range_start approves
         * a non-blocking behavior then the same applies to
         * invalidate_range_end.
         *
         */
        int (*invalidate_range_start)(struct mmu_notifier *mn,
                                      const struct mmu_notifier_range *range);
        void (*invalidate_range_end)(struct mmu_notifier *mn,
                                     const struct mmu_notifier_range *range);

        /*
         * invalidate_range() is either called between
         * invalidate_range_start() and invalidate_range_end() when the
         * VM has to free pages that where unmapped, but before the
         * pages are actually freed, or outside of _start()/_end() when
         * a (remote) TLB is necessary.
         *
         * If invalidate_range() is used to manage a non-CPU TLB with
         * shared page-tables, it not necessary to implement the
         * invalidate_range_start()/end() notifiers, as
         * invalidate_range() alread catches the points in time when an
         * external TLB range needs to be flushed. For more in depth
         * discussion on this see Documentation/vm/mmu_notifier.rst
         *
         * Note that this function might be called with just a sub-range
         * of what was passed to invalidate_range_start()/end(), if
         * called between those functions.
         */
        void (*invalidate_range)(struct mmu_notifier *mn, struct mm_struct *mm,
                                 unsigned long start, unsigned long end);
};
```

- (*invalidate_range_start)
- (*invalidate_range_end)
  - The above two hook functions work in pairs. When invalidating TLB for VMs for that range of primary MMU, the secondary MMU also performs TLB invalidation, calling these hook functions before and after the routine of manipulating them en masse via rmap walk.
- (*invalidate_range)
  - After invalidating the given scope of the primary MMU in the VM, the secondary MMU is also called as well.

In the following figure, you can see the position of init, start, end, etc. for the secondary MMU when using the MMU notifier.


mmu_notifier_range_init()
mmu_notifier_invalidate_range_start()
여러 개의 매핑 조작
set_pte()
for primary MMU
mmu_notifier_invalidate_range()
for secondary MMU
mmu_notifier_invalidate_range_end()
(http://jake.dothome.co.kr/wp-content/uploads/2019/09/mmu-notifier-2a.png)

MM Counter

- RSS(Resident Set Size)
  - The process is mapped to the actual physical memory and refers to the number of pages in use.
  - Except for swapped out pages, this includes stack and heap memory.

- VSZ(Virutal memory SiZe)
  - The process is the number of virtual address pages being used.
  - It includes pages that are larger than RSS and have been swapped out, as well as pages that have been allocated space but have never been accessed and therefore have no physical memory mapped to them.

You can see the RSS and VSZ counters used by the bash (pid=5831) process as follows:

```
$ cat /proc/5831/status
Name:   bash
State:  S (sleeping)
Tgid:   5831
Ngid:   0
Pid:    5831
PPid:   5795
TracerPid:      0
Uid:    0       0       0       0
Gid:    0       0       0       0
FDSize: 256
Groups: 0
NStgid: 5831
NSpid:  5831
NSpgid: 5831
NSsid:  5831
VmPeak:     6768 kB     <--- hiwater VSZ
VmSize:     6704 kB     <--- VSZ
VmLck:         0 kB
VmPin:         0 kB
VmHWM:      4552 kB     <--- hiwater RSS (file + anon + shm)
VmRSS:      3868 kB     <--- RSS
VmData:     1652 kB
VmStk:       132 kB
VmExe:       944 kB
VmLib:      1648 kB
VmPTE:        28 kB
VmPMD:        12 kB
VmSwap:        0 kB
Threads:        1
SigQ:   0/15244
SigPnd: 0000000000000000
ShdPnd: 0000000000000000
SigBlk: 0000000000000000
SigIgn: 0000000000380004
SigCgt: 000000004b817efb
CapInh: 0000000000000000
CapPrm: 0000003fffffffff
CapEff: 0000003fffffffff
CapBnd: 0000003fffffffff
CapAmb: 0000000000000000
Seccomp:        0
Cpus_allowed:   3f
Cpus_allowed_list:      0-5
Mems_allowed:   1
Mems_allowed_list:      0
voluntary_ctxt_switches:        50
nonvoluntary_ctxt_switches:     29
```

Check page references

page_referenced()

mm/rmap.c

```
/**
 * page_referenced - test if the page was referenced
 * @page: the page to test
 * @is_locked: caller holds lock on the page
 * @memcg: target memory cgroup
 * @vm_flags: collect encountered vma->vm_flags who actually referenced
the page
 *
 * Quick test_and_clear_referenced for all mappings to a page,
 * returns the number of ptes which referenced the page.
 */
```

```
int page_referenced(struct page *page,
                    int is_locked,
                    struct mem_cgroup *memcg,
                    unsigned long *vm_flags)
{
        int we_locked = 0;
        struct page_referenced_arg pra = {
                .mapcount = total_mapcount(page),
                .memcg = memcg,
        };
        struct rmap_walk_control rwc = {
                .rmap_one = page_referenced_one,
                .arg = (void *)&pra,
                .anon_lock = page_lock_anon_vma_read,
        };

        *vm_flags = 0;
        if (!page_mapped(page))
                return 0;

        if (!page_rmapping(page))
                return 0;

        if (!is_locked && (!PageAnon(page) || PageKsm(page))) {
                we_locked = trylock_page(page);
                if (!we_locked)
                        return 1;
        }

        /*
         * If we are reclaiming on behalf of a cgroup, skip
         * counting on behalf of references from different
         * cgroups
         */
        if (memcg) {
                rwc.invalid_vma = invalid_page_referenced_vma;
        }

        rmap_walk(page, &rwc);
        *vm_flags = pra.vm_flags;

        if (we_locked)
                unlock_page(page);

        return pra.referenced;
}
```

page_referenced_one()

mm/rmap.c

```
/*
 * arg: page_referenced_arg will be passed
 */
```

```
static bool page_referenced_one(struct page *page, struct vm_area_struct *vma,
                        unsigned long address, void *arg)
{
        struct page_referenced_arg *pra = arg;
        struct page_vma_mapped_walk pvmw = {
                .page = page,
                .vma = vma,
                .address = address,
        };
        int referenced = 0;

        while (page_vma_mapped_walk(&pvmw)) {
                address = pvmw.address;

                if (vma->vm_flags & VM_LOCKED) {
                        page_vma_mapped_walk_done(&pvmw);
                        pra->vm_flags |= VM_LOCKED;
                        return false; /* To break the loop */
                }

                if (pvmw.pte) {
                        if (ptep_clear_flush_young_notify(vma, address,
                                                pvmw.pte)) {
                                /*
                                 * Don't treat a reference through
                                 * a sequentially read mapping as such.
                                 * If the page has been used in another mapping,
                                 * we will catch it; if this other mapping is
                                 * already gone, the unmap path will have set
                                 * PG_referenced or activated the page.
                                 */
                                if (likely(!(vma->vm_flags & VM_SEQ_READ)))
                                        referenced++;
                        }
                } else if (IS_ENABLED(CONFIG_TRANSPARENT_HUGEPAGE)) {
                        if (pmdp_clear_flush_young_notify(vma, address,
                                                pvmw.pmd))
                                referenced++;
                } else {
                        /* unexpected pmd-mapped page? */
                        WARN_ON_ONCE(1);
                }

                pra->mapcount--;
        }

        if (referenced)
                clear_page_idle(page);
        if (test_and_clear_page_young(page))
                referenced++;

        if (referenced) {
                pra->referenced++;
                pra->vm_flags |= vma->vm_flags;
        }

        if (!pra->mapcount)
                return false; /* To break the loop */

        return true;
}
```

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) | Sentence C – Current post
- Rmap -3- (PVMW) (http://jake.dothome.co.kr/rmap-3) | Qc

LEAVE A COMMENT

Your email will not be published. Required fields are marked with *

Comments

name *

email *

Website

WRITE A COMMENT

Rmap -3- (PVMW) (http://jake.dothome.co.kr/rmap-3/)

Rmap -1- (Reverse Mapping) (http://jake.dothome.co.kr/rmap-1/)

Munc Blog (2015 ~ 2023)