Rmap -1- (Reverse Mapping)

2019-09-10 (http://jake.dothome.co.kr/rmap-1/) Moon Young-il (http://jake.dothome.co.kr/author/admin/) Linux Kernel (http://jake.dothome.co.kr/category/linux/)

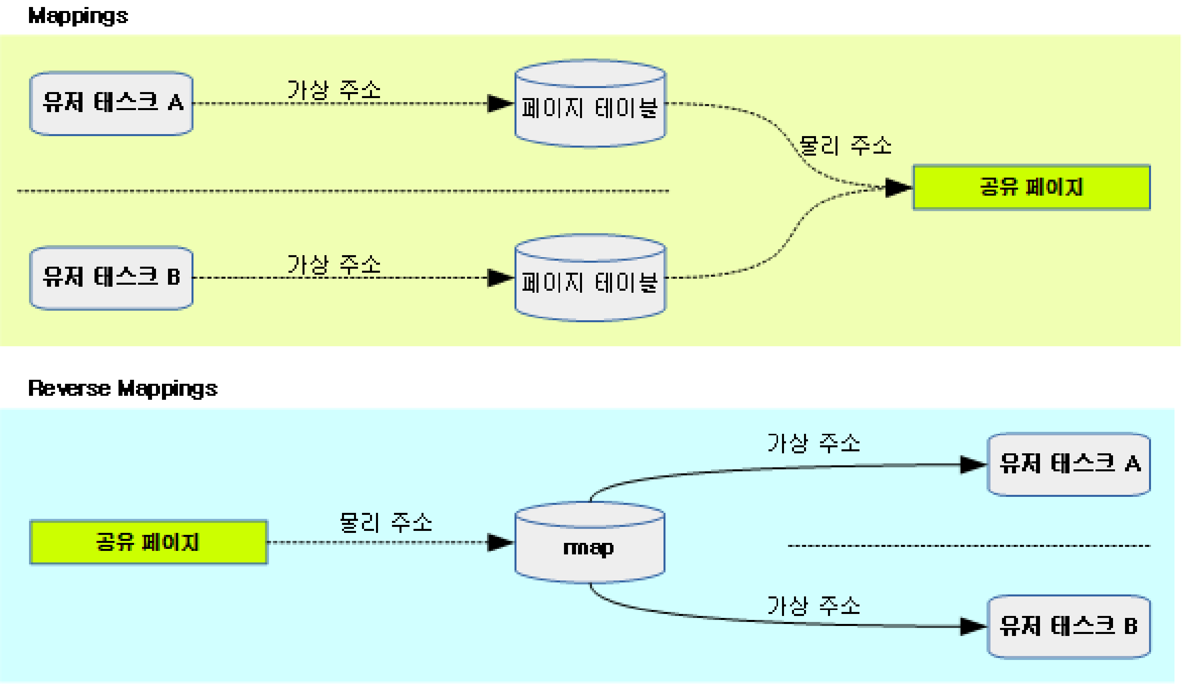
<kernel v5.15>

Rmap -1- (Reverse Mapping)

Rmap is a method of using a physical address and backmapping it to a virtual address. The API uses this rmap to rmap_walk find all VMs and virtual addresses that use the physical page, and asks you to perform various actions on the mappings (VMs, virtual addresses, pte) that you find.

- try_to_unmap()
- page_mkclean()
- page_referenced()
- try_to_migrate()
- page_mlock()
- page_make_device_exclusive()
- remove_migration_ptes()
- page_idle_clear_pte_refs()
- try_to_munlock()

The following illustration illustrates the concept of rmap using forward and reverse mapping.

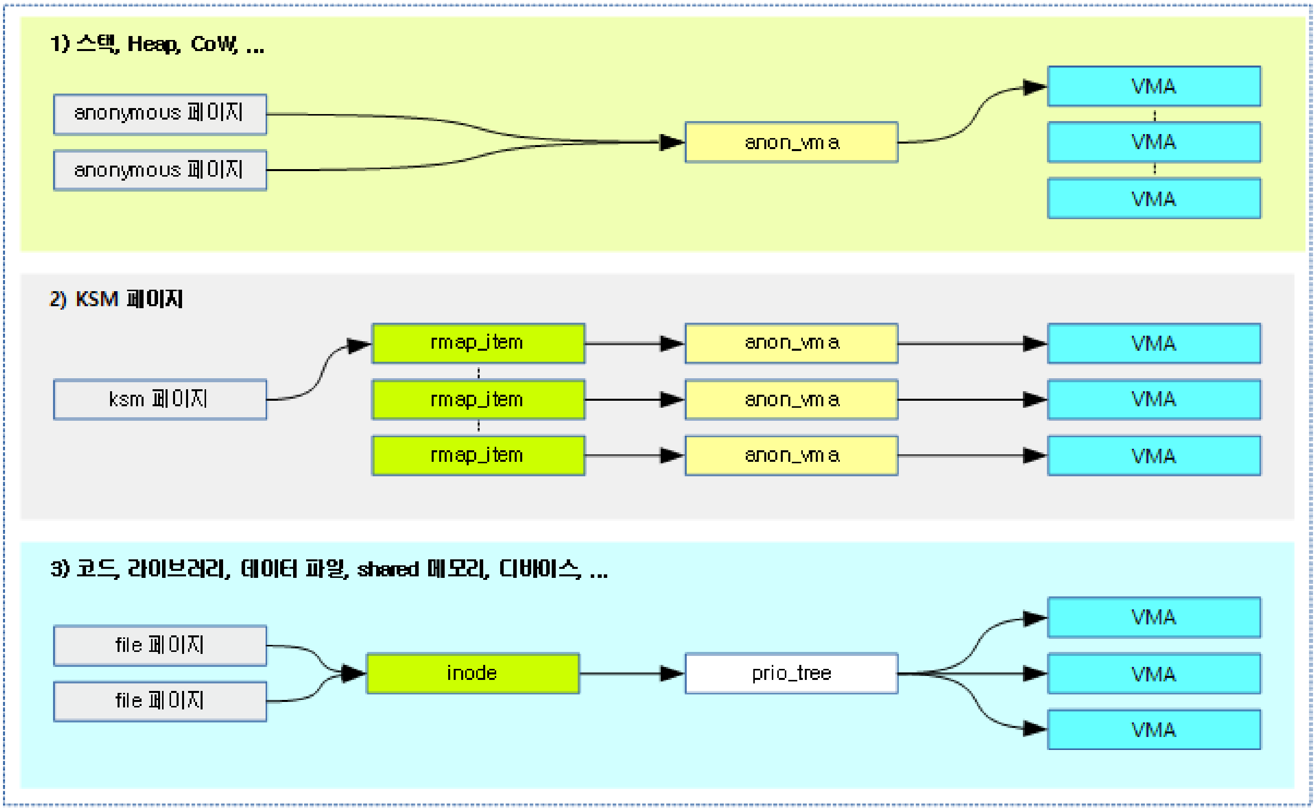


(http://jake.dothome.co.kr/wp-content/uploads/2019/09/rmap-1.png)

rmap type

As shown in the following figure, there are four different types of rmap configurations depending on the user page type.

- Anonymous Mapping
    - It is used when user stacks, user heaps, and CoW pages are allocated.
    - No. 1) in the figure below
- KSM Mapping
    - In the case of VM_MERGEABLE regions, user anon pages can be merged if the physical memory data is the same.
    - No. 2) in the figure below
- File Mapping
    - It is used to load and allocate code, libraries, data files, shared memory, devices, etc.
    - No. 3) in the figure below
- Non-LRU Movable Mapping
    - It is used to map pages on file systems that support non-LRU movables, and these pages can be migrated.
    - No. 3) in the figure below
    - It was added in kernel v2016.4-rc8 in 1.
        - Note: MM: Migrate: Support Non-LRU Movable Page Migration (https://github.com/torvalds/linux/commit/bda807d4445414e8e77da704f116bb0880fe0c76) (2016, v4.8-rc1)



(http://jake.dothome.co.kr/wp-content/uploads/2019/09/rmap-5a.png)

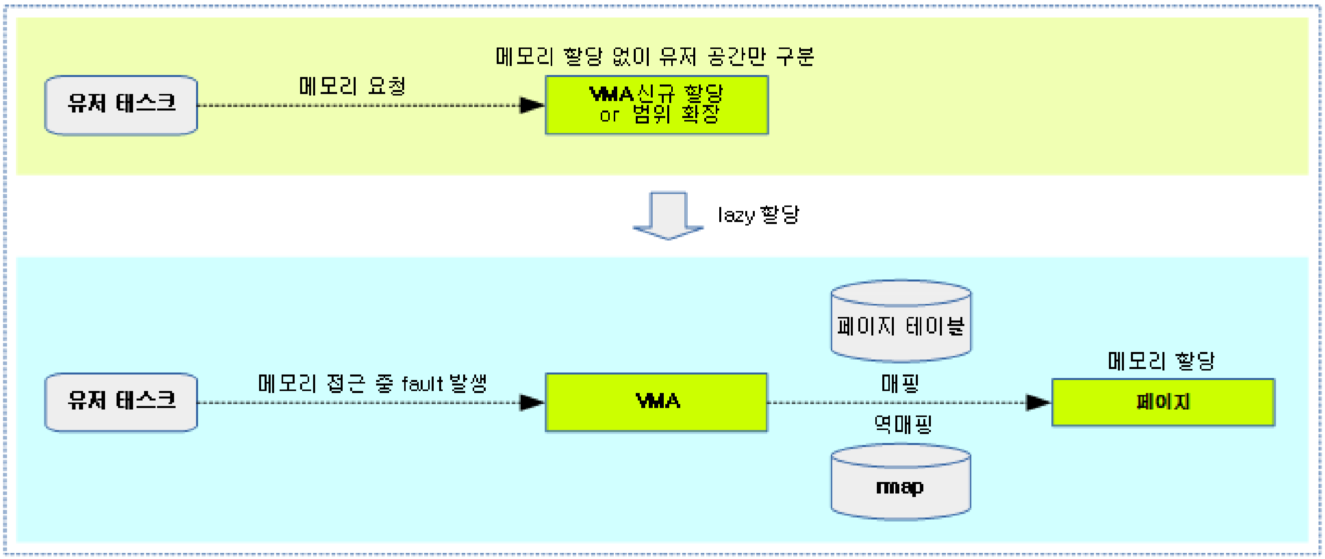
The mapping member of the page structure points to the above structs, and can be distinguished by flags in the lower 2 bits.

- AnyNyMouse Page
    - PAGE_MAPPING_ANON(1)
- KSM Pages
    - PAGE_MAPPING_KSM(3) = PAGE_MAPPING_ANON(1) + PAGE_MAPPING_MOVABLE(2)
- file page
    - Lower bits all 0
- Non-LRU Movable Pages
    - PAGE_MAPPING_MOVABLE(2)

Creating Fault Handlers and VMAs

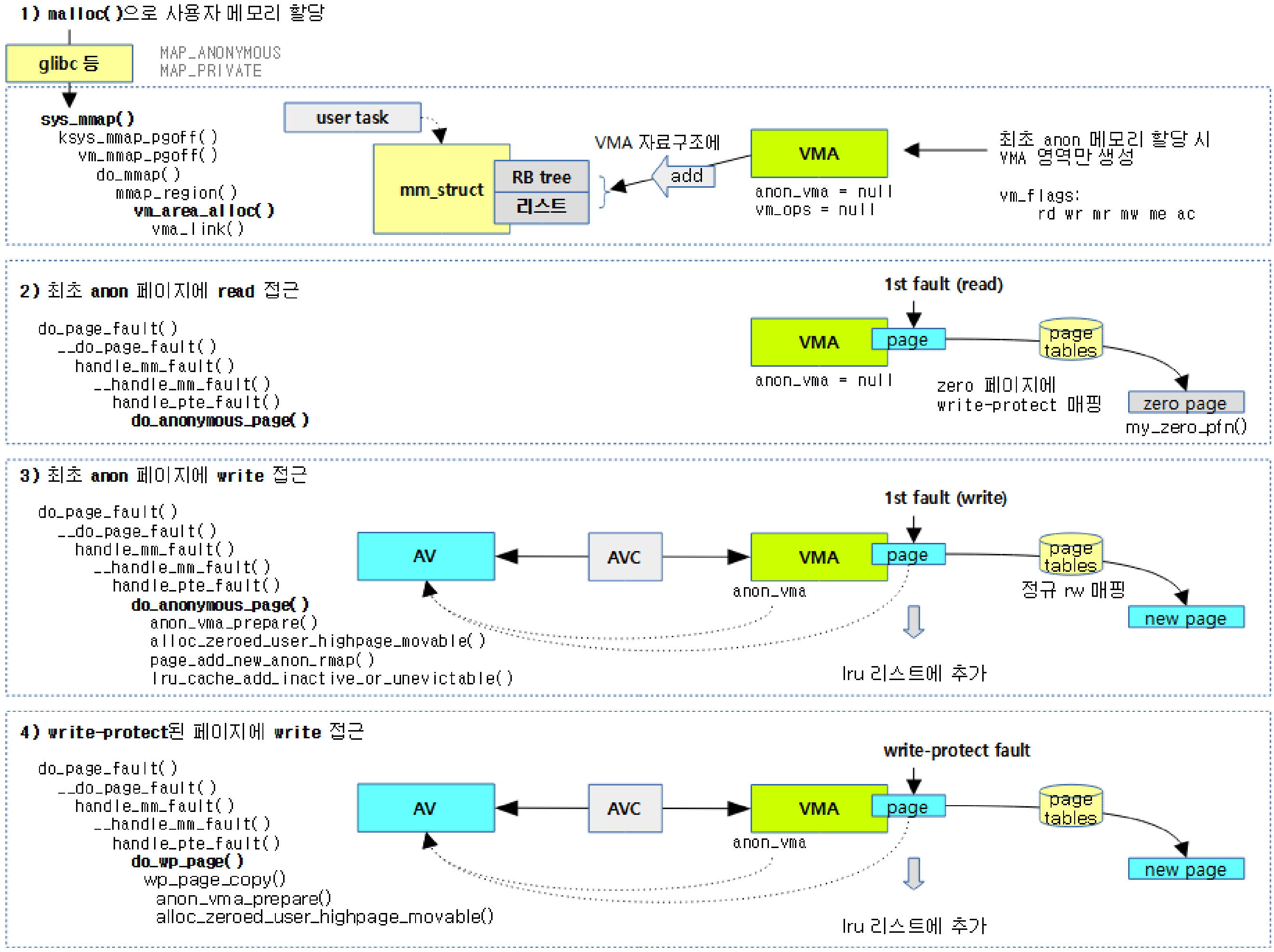
When user memory allocation is requested, the kernel creates or extends VMAs with a mark that it uses virtual address space, and does not allocate physical memory. Then, when accessing the user's area, a fault occurs in which VMA the fault handler knows which VMA the fault address is located in. And if the faulted address is valid, it uses a lazy allocation policy that belatedly allocates a physical page. At this point, we use forward mapping to update the page table entry, and we also add rmap, which is a reverse mapping.

The following figure shows the process by which page allocation and mapping/backmapping are performed by the fault handler when using physical memory after a page allocation request.



(http://jake.dothome.co.kr/wp-content/uploads/2019/09/rmap-2a.png)

The following figure shows how the first VMA is created using the fault handler.



```
sys_mmap()
  ksys_mmap_pgoff()
    vm_mmap_pgoff()
      do_mmap()
        mmap_region()
          vm_area_alloc()
            vma_link()
```

```
do_page_fault()
  __do_page_fault()
    handle_mm_fault()
      __handle_mm_fault()
        handle_pte_fault()
          do_anonymous_page()
```

```
do_page_fault()
  __do_page_fault()
    handle_mm_fault()
      __handle_mm_fault()
        handle_pte_fault()
          do_anonymous_page()
            anon_vma_prepare()
            alloc_zeroed_user_highpage_movable()
            page_add_new_anon_rmap()
            lru_cache_add_inactive_or_unevictable()
```

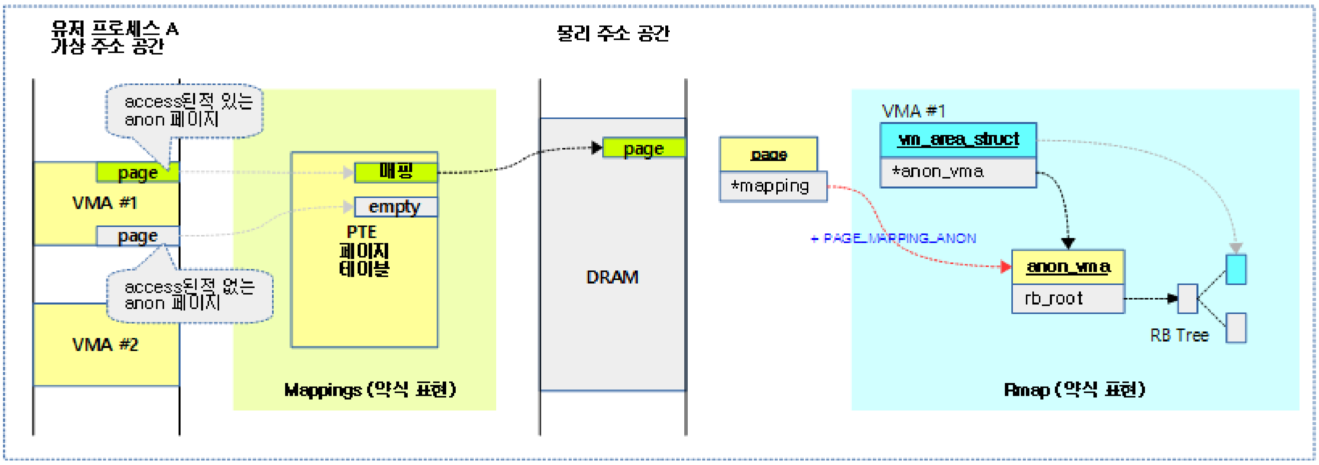
```
do_page_fault()
  __do_page_fault()
    handle_mm_fault()
      __handle_mm_fault()
        handle_pte_fault()
          do_wp_page()
           wp_page_copy()
            anon_vma_prepare()
            alloc_zeroed_user_highpage_movable()
```

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

Creating a Fault Handler and an Rmap

The following figure shows an overview of the anon rmap status.

- For forward mapping, use a multi-step page table (pgd -> p4d -> pud -> pmd -> pte).
- In reverse mapping, each anon page is represented by a anon_vma structure (AV).
  - In the figure below, VMA and AV are connected via AVC (anon_vma_chain), but the AVC notation is omitted.
  - When linking, page->mapping uses the anon_vma struct address with the PAGE_MAPPING_ANON flag added.



(http://jake.dothome.co.kr/wp-content/uploads/2019/09/rmap-3b.png)

Virtual Address Space Management (VMA Management)

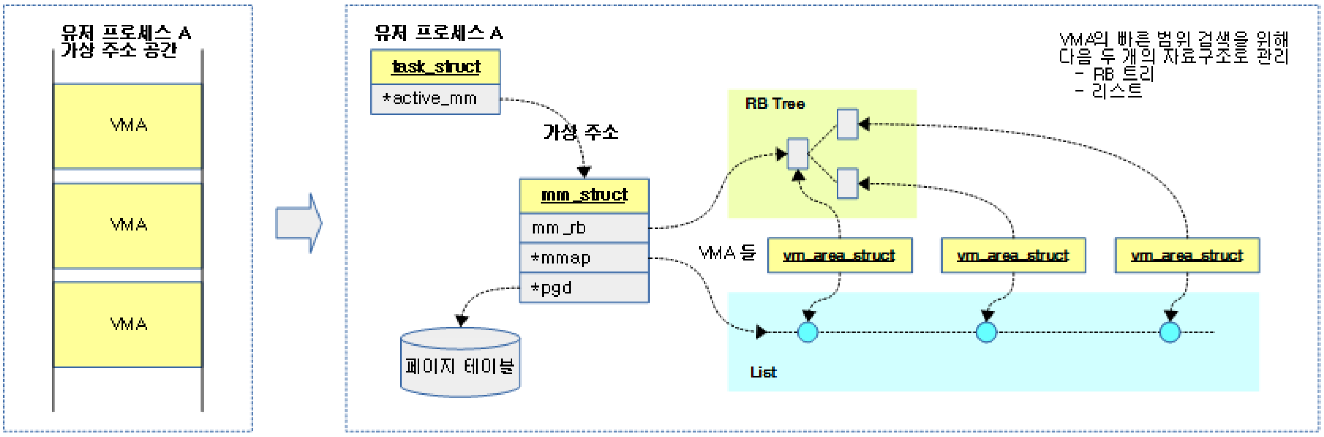
Each user process has a virtual address space, and a page table is used to map them. In the user virtual address space, several virtual address zones requested by the user are registered, and these are represented by assigning vm_area_struct structures to the VMA.

- Note that if there are child threads in the process, the child threads are also managed as task_struct. However, the user address space is shared and used by the user process space. Therefore, the mm_struct of all child threads is the same.

The data structures managed by VMA are managed using two data structures, the RB tree and the list, and the reason for using the two is that they are used to effectively respond to empty space range searches.

- RB Tree
  - You can quickly respond to searches for the starting address.
- Linear LinkedIn Lists
  - VMAs are sorted by address, so you can quickly find out the free space size between the aligned VMAs.

The following figure shows three virtual address zones (VMAs) registered and managed in the user's address space.



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

The VMAs registered in the process (pid 1481) are shown as follows: Check for Anon VMA that is not related to the file.

- They are displayed in the following order:
  - Start Virtual Address (VMA->vm_start)
  - End Virtual Address (VMA->vm_end)
  - Flag (VM_READ, VM_WRITE, VM_EXEC, VM_MAYSHARE)
  - Offset address if the file is mapped (VMA->vm_pgoff << PAGE_SHIFT)
  - If the file is mapped, the device's major and minor numbers
  - If the file is mapped, the inode number
  - File name if the file is mapped, or with a space or parenthesis [] for Anon VMA
    - The Heap, VVAR, VDSO, and Stack zones are specifically ANON VMAs, represented by parentheses [].

```
$ cat /proc/1481/maps
00400000-004ec000 r-xp 00000000 b3:05 26                                 /bin/bash
004fb000-004ff000 r--p 000eb000 b3:05 26                                 /bin/bash
004ff000-00508000 rw-p 000ef000 b3:05 26                                 /bin/bash
00508000-00512000 rw-p 00000000 00:00 0
30d7f000-30f1d000 rw-p 00000000 00:00 0                                  [heap]
7f87324000-7f8732d000 r-xp 00000000 b3:05 2173                           /lib/aarch
64-linux-gnu/libnss_files-2.24.so
7f8732d000-7f8733c000 ---p 00009000 b3:05 2173                           /lib/aarch
64-linux-gnu/libnss_files-2.24.so
7f8733c000-7f8733d000 r--p 00008000 b3:05 2173                           /lib/aarch
64-linux-gnu/libnss_files-2.24.so
7f8733d000-7f8733e000 rw-p 00009000 b3:05 2173                           /lib/aarch
64-linux-gnu/libnss_files-2.24.so
7f8733e000-7f87344000 rw-p 00000000 00:00 0
7f87344000-7f8734d000 r-xp 00000000 b3:05 2197                           /lib/aarch
64-linux-gnu/libnss_nis-2.24.so
7f87370000-7f8737f000 ---p 00012000 b3:05 2243                           /lib/aarch
64-linux-gnu/libnsl-2.24.so
(...생략...)
7f8771b000-7f8771c000 r--p 00000000 b3:05 7362                           /usr/lib/l
ocale/C.UTF-8/LC_IDENTIFICATION
7f8771c000-7f87721000 rw-p 00000000 00:00 0
7f87721000-7f87722000 r--p 00000000 00:00 0                              [vvar]
7f87722000-7f87723000 r-xp 00000000 00:00 0                              [vdso]
7f87723000-7f87724000 r--p 0001c000 b3:05 2253                           /lib/aarch
64-linux-gnu/ld-2.24.so
7f87724000-7f87726000 rw-p 0001d000 b3:05 2253                           /lib/aarch
64-linux-gnu/ld-2.24.so
7fcdc27000-7fcdc48000 rw-p 00000000 00:00 0                              [stack]
```

You can use the following command to view the above information in more detail.

```
$ cat /proc/1481/smaps
00400000-004ec000 r-xp 00000000 b3:05 26                                 /bin/bash
Size:                944 kB
Rss:                 936 kB
Pss:                 311 kB
Shared_Clean:        936 kB
Shared_Dirty:          0 kB
Private_Clean:         0 kB
Private_Dirty:         0 kB
Referenced:          936 kB
Anonymous:             0 kB
AnonHugePages:         0 kB
Shared_Hugetlb:        0 kB
Private_Hugetlb:       0 kB
Swap:                  0 kB
SwapPss:               0 kB
KernelPageSize:        4 kB
MMUPageSize:           4 kB
Locked:                0 kB
VmFlags: rd ex mr mw me dw
(...생략...)
7fcdc27000-7fcdc48000 rw-p 00000000 00:00 0                              [stack]
Size:                132 kB
Rss:                  32 kB
Pss:                  32 kB
Shared_Clean:          0 kB
Shared_Dirty:          0 kB
Private_Clean:         0 kB
Private_Dirty:        32 kB
Referenced:           32 kB
Anonymous:            32 kB
AnonHugePages:         0 kB
Shared_Hugetlb:        0 kB
Private_Hugetlb:       0 kB
Swap:                  0 kB
SwapPss:               0 kB
KernelPageSize:        4 kB
MMUPageSize:           4 kB
Locked:                0 kB
VmFlags: rd wr mr mw me gd ac
```

VM Flags

The following are the VM flags used in vma->vm_flags:

- The two letters on the right are maps or smaps, which is an abbreviated notation when printed.

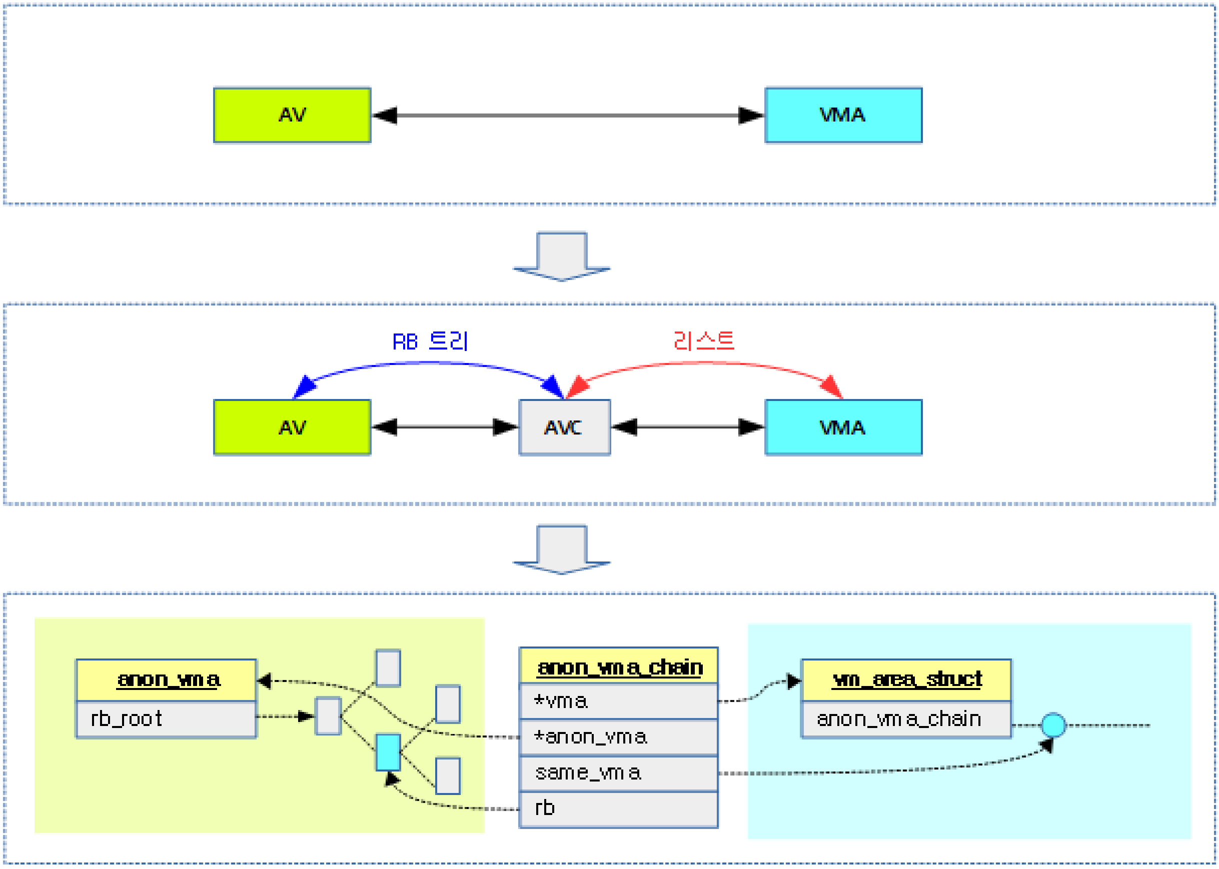
```
VM_READ         rd
VM_WRITE        wr
VM_EXEC         ex
VM_SHARED       sh
VM_MAYREAD      mr
VM_MAYWRITE     mw
VM_MAYEXEC      me
VM_MAYSHARE     ms
VM_GROWSDOWN    gd
VM_UFFD_MISSING
VM_PFNMAP       pf
VM_DENYWRITE    dw
VM_UFFD_WP
VM_LOCKED       lo
VM_IO           io
VM_SEQ_READ     sr
VM_RAND_READ    rr
VM_DONTCOPY     dc
VM_DONTEXPAND   de
VM_LOCKONFAULT
VM_ACCOUNT      ac
VM_NORESERVE    nr
VM_HUGETLB      ht
VM_SYNC         sf
VM_ARCH_1       ar
VM_WIPEONFORK   wf
VM_DONTDUMP     dd
VM_SOFTDIRTY    sd
VM_MIXEDMAP     mm
VM_HUGEPAGE     hg
VM_NOHUGEPAGE   nh
VM_MERGEABLE    mg
```

anonymous type VMA

anon_vma structure (AV) is used to manage the anonymous type VMA(vm_area_struct), and AVC (anon_vma_chain structure) is used to manage the connection between the VMA and the AV.

- AVs are managed using RB trees to include many VMAs.
    - In September 2012, kernel v9.3-rc7 replaced the linear linked list used for AV with an interval tree using RB tree.
    - Note: mm anon rmap: replace same_anon_vma linked list with an interval tree. (https://github.com/torvalds/linux/commit/bf181b9f9d8dfbba58b23441ad60d0bc33806d64#diff-82d4b79a51478b4ef923b8d2f2ffdee6)
- The VMA still manages it using a linear LinkedIn list to include AVs.

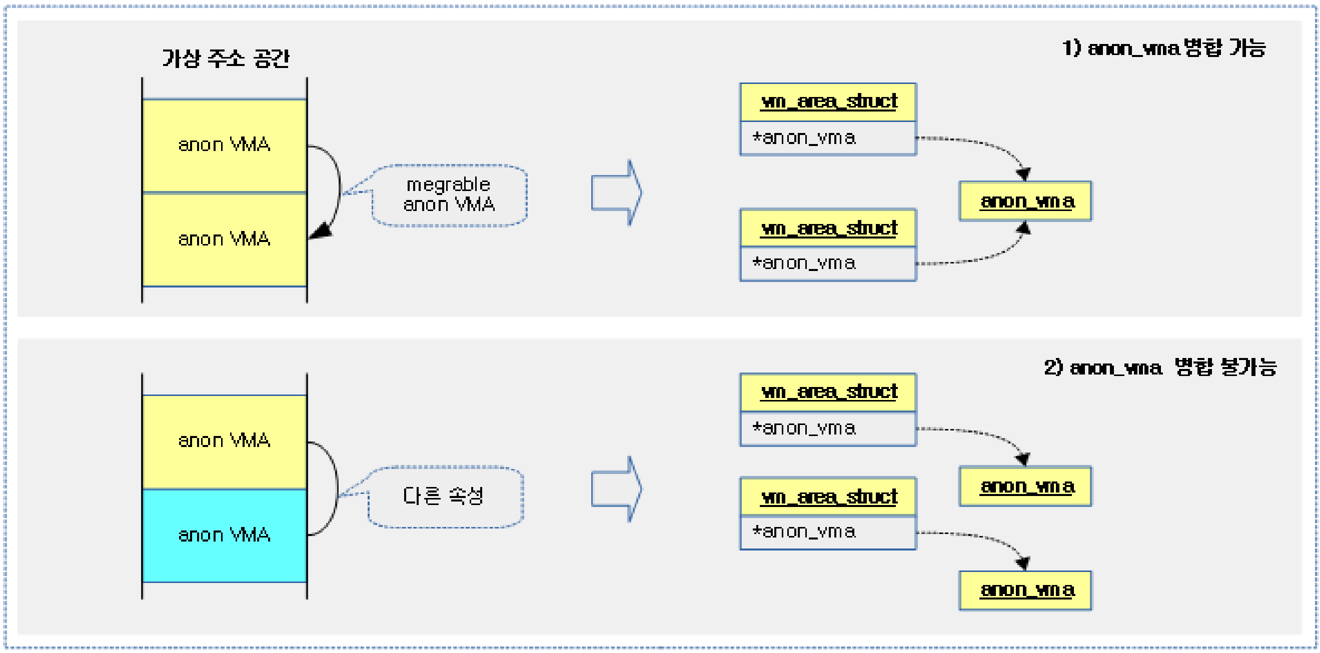
The following figure shows AVC being used for the connection between VMA and AV.



(http://jake.dothome.co.kr/wp-content/uploads/2019/09/rmap-4a.png)

Merge anon_vma

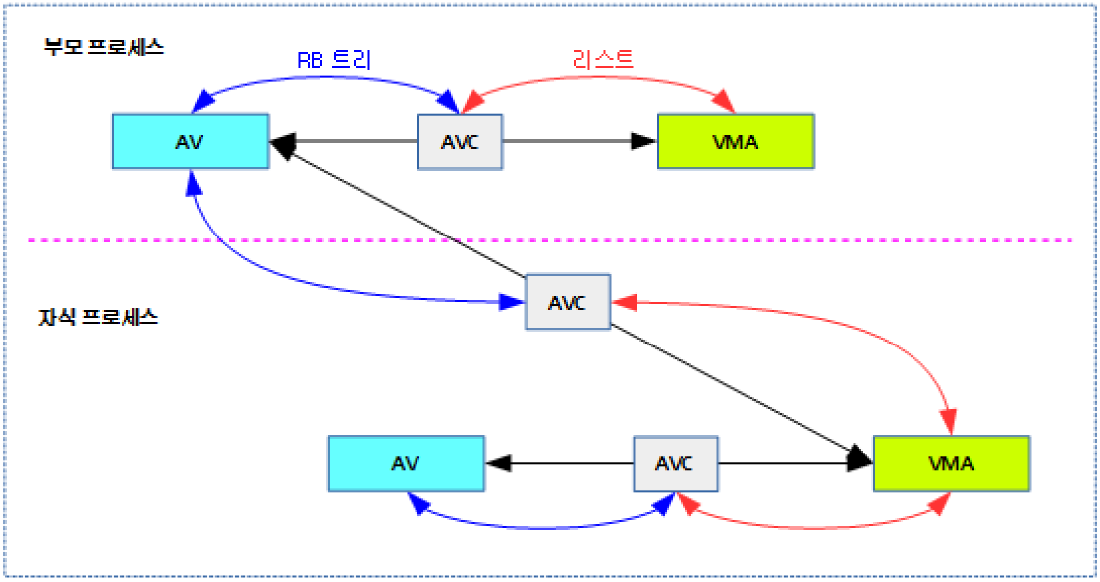
If the VMAs are adjacent and have similar properties, you can use the existing one without creating a separate anon_vma.



(http://jake.dothome.co.kr/wp-content/uploads/2019/09/rmap-8.png)

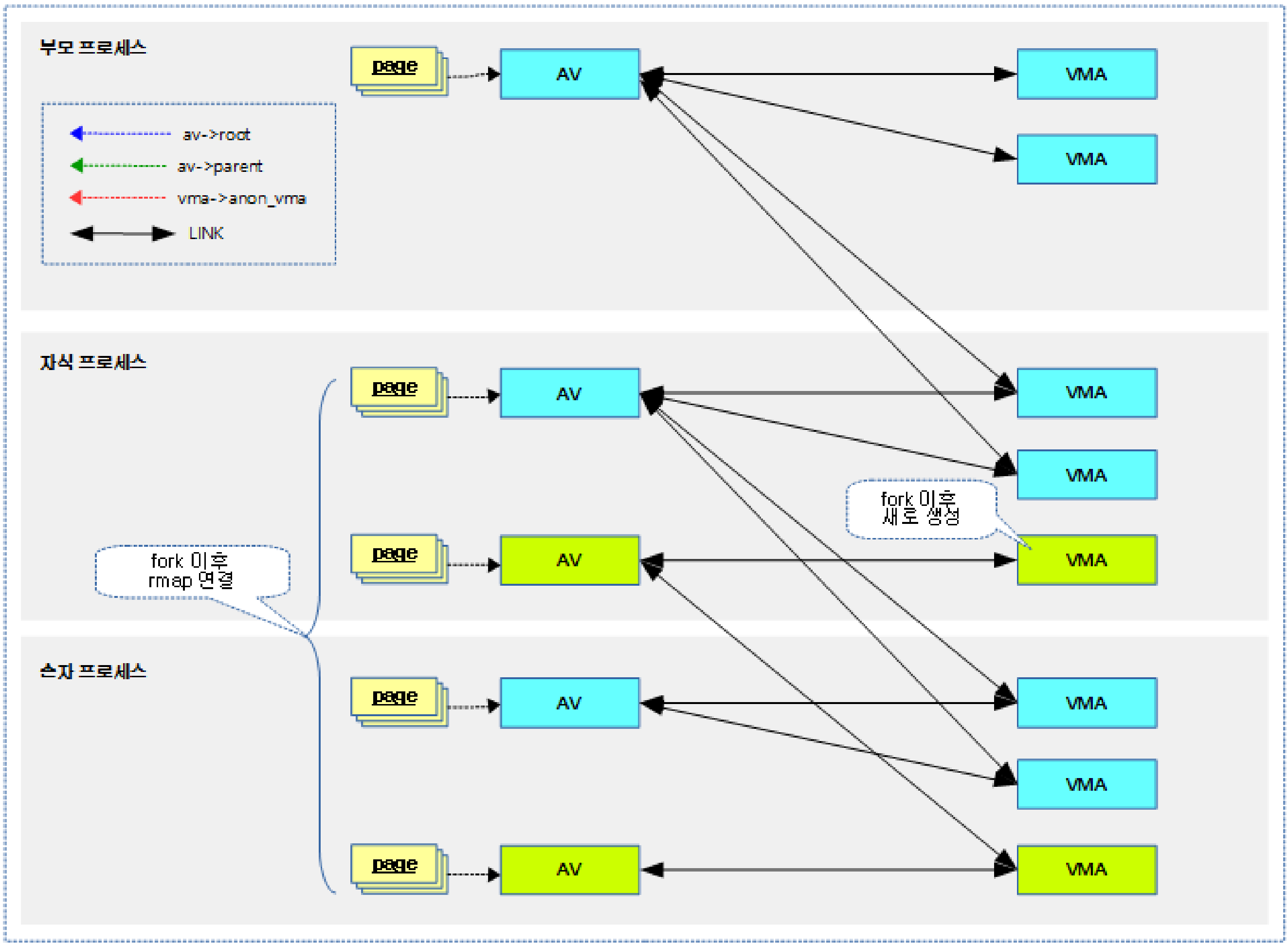
Management in the Forked Child Process

The following figure shows the relationship to the forked child process.



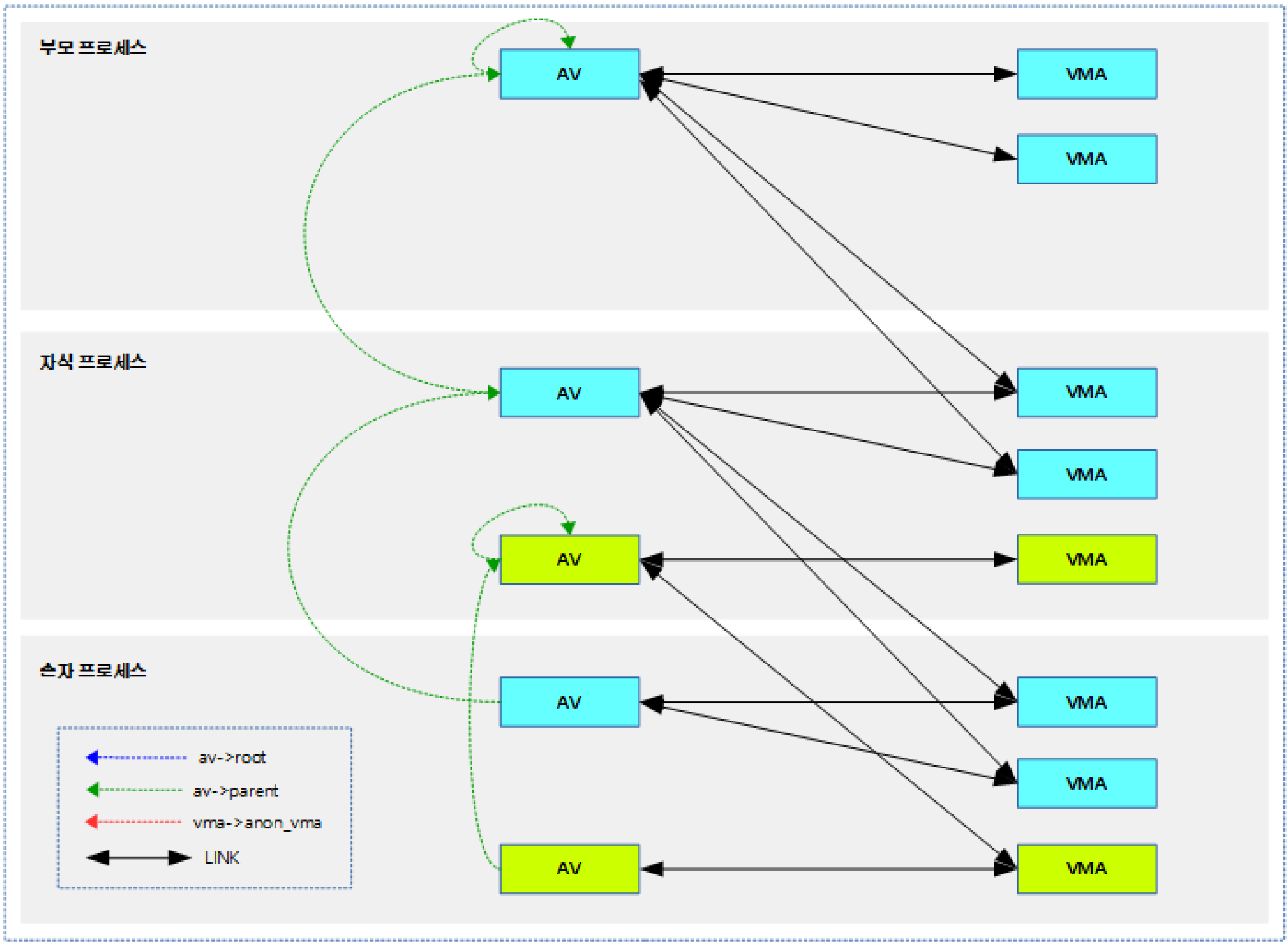
(http://jake.dothome.co.kr/wp-content/uploads/2019/09/rmap-6a.png)

The following figure shows how the parent VMA is cloneked and the AV is created and linked by forking the two child processes.



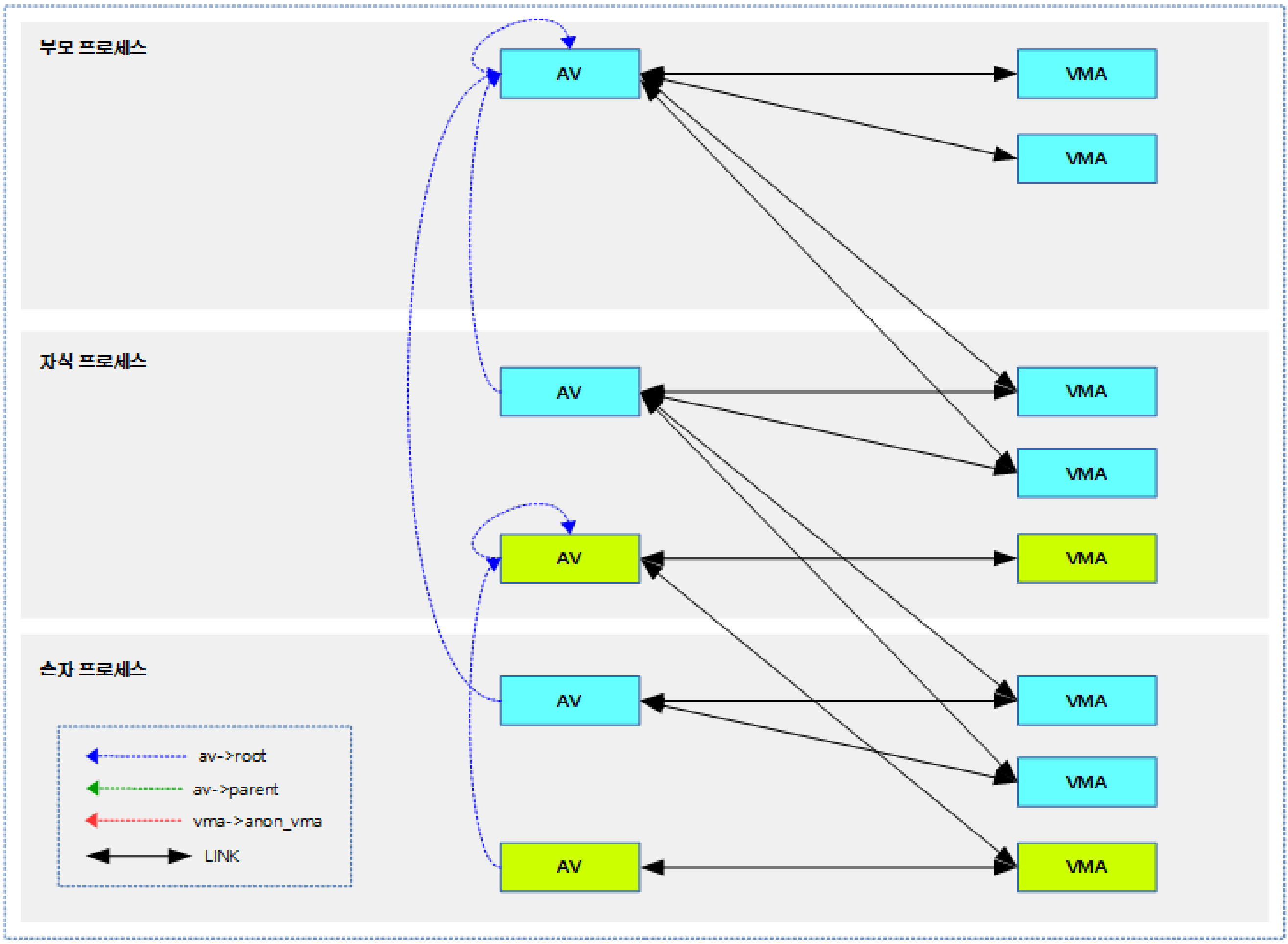
(http://jake.dothome.co.kr/wp-content/uploads/2019/09/rmap-9b.png)

The parent relationship of AV is represented as shown in the following figure.



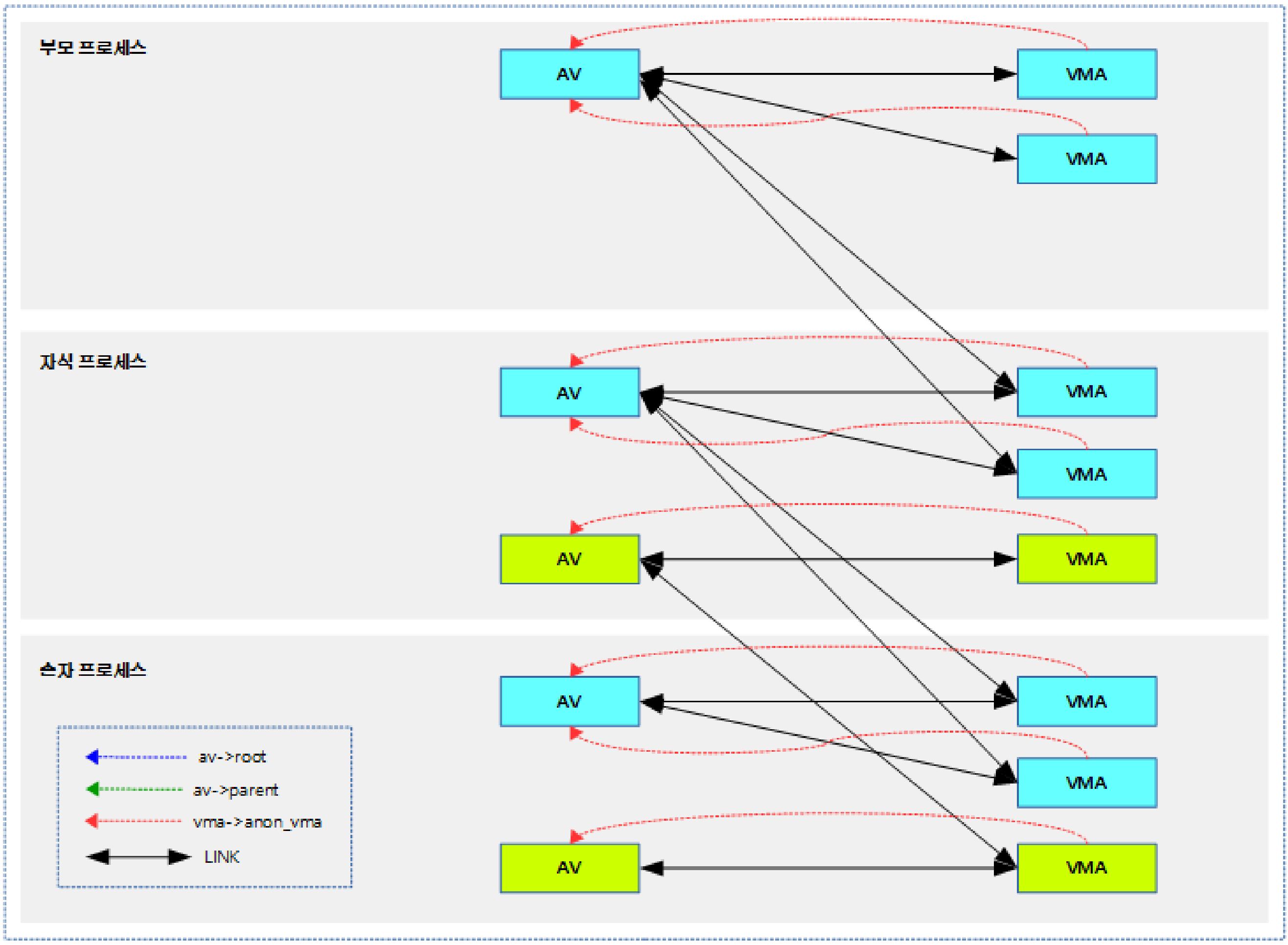
(http://jake.dothome.co.kr/wp-content/uploads/2019/09/rmap-10a.png)

The first generated root relationship of the AV is represented in the following figure.



(http://jake.dothome.co.kr/wp-content/uploads/2019/09/rmap-11.png)

As shown in the following figure, the relationship that VMA points to directly 1:1 is expressed.



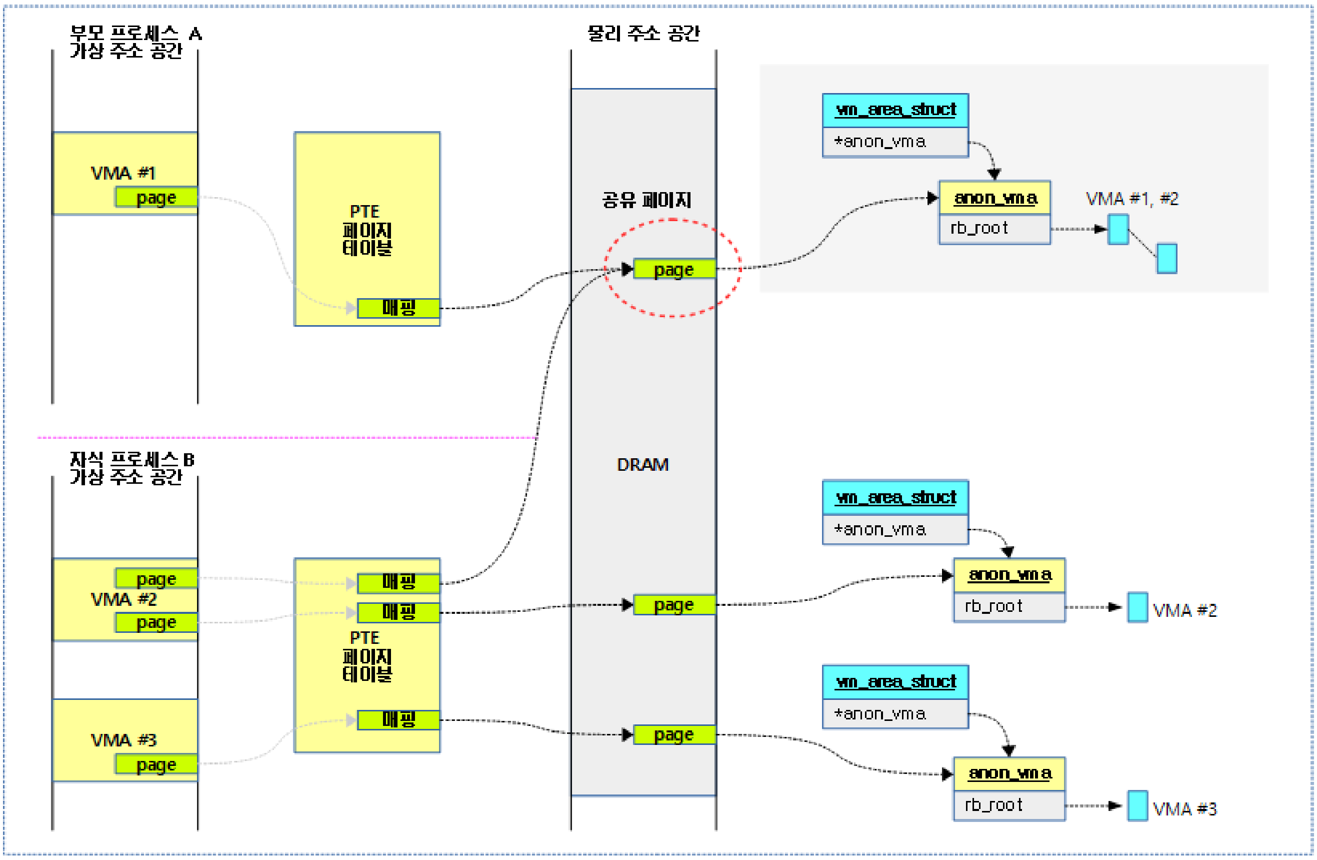
(http://jake.dothome.co.kr/wp-content/uploads/2019/09/rmap-12.png)

Efficient mapping elimination with rmap

When you remove a mapping from a shared page, you have to dig through all the numerous user page tables used for forward mappings to use only forward mappings. You can use reverse mapping to find the virtual address zone (VMA) and unmap it only from the user page table used by the VMA to get rid of it quickly.

The following figure shows how VMA #1, #2를, is managed to find the VMA #<>, to search for, when removing the mapping of a shared page between the parent process and the forked child process.

- When child process B is forked, it clone VMA#1 to create a VMA #2를.
- You can see that the newly assigned pages of child process B are created and managed in VMA#2 and VMA#3, respectively.



(http://jake.dothome.co.kr/wp-content/uploads/2019/09/rmap-7b.png)

anon_vma Initialization

anon_vma_init()

mm/rmap.c

```
1  void __init anon_vma_init(void)
2  {
3          anon_vma_cachep = kmem_cache_create("anon_vma", sizeof(struct anon_vma),
4                          0, SLAB_TYPESAFE_BY_RCU|SLAB_PANIC|SLAB_ACCOUNT,
5                          anon_vma_ctor);
6          anon_vma_chain_cachep = KMEM_CACHE(anon_vma_chain,
7                          SLAB_PANIC|SLAB_ACCOUNT);
8  }
```

Create a slub cache for the purpose of allocating anon_vma and anon_vma_chain structs.

- In line 3 of code, create a slub cache for the purpose of allocating anon_vma structs, and have the anon_vma_ctor() function called at initialization.
- In line 6 of code, create a slub cache for the purpose of allocating anon_vma_chain structs.

anon_vma Assignment

anon_vma_alloc()

mm/rmap.c

```
static inline struct anon_vma *anon_vma_alloc(void)
{
        struct anon_vma *anon_vma;

        anon_vma = kmem_cache_alloc(anon_vma_cachep, GFP_KERNEL);
        if (anon_vma) {
                atomic_set(&anon_vma->refcount, 1);
                anon_vma->degree = 1;   /* Reference for first vma */
                anon_vma->parent = anon_vma;
                /*
                 * Initialise the anon_vma root to point to itself. If called
                 * from fork, the root will be reset to the parents anon_vma.
                 */
                anon_vma->root = anon_vma;
        }

        return anon_vma;
}
```

Assign anon_vma.

- Set the degree of the member of the assigned anon_vma structure to 1 to identify the one at the foremost, and also point the parent and root to themselves.

Turn off anon_vma

anon_vma_free()

mm/rmap.c

```
static inline void anon_vma_free(struct anon_vma *anon_vma)
{
        VM_BUG_ON(atomic_read(&anon_vma->refcount));

        /*
         * Synchronize against page_lock_anon_vma_read() such that
         * we can safely hold the lock without the anon_vma getting
         * freed.
         *
         * Relies on the full mb implied by the atomic_dec_and_test() from
         * put_anon_vma() against the acquire barrier implied by
         * down_read_trylock() from page_lock_anon_vma_read(). This orders:
         *
         * page_lock_anon_vma_read()    VS      put_anon_vma()
         *   down_read_trylock()                  atomic_dec_and_test()
         *   LOCK                                 MB
         *   atomic_read()                        rwsem_is_locked()
         *
         * LOCK should suffice since the actual taking of the lock must
         * happen _before_ what follows.
         */
        might_sleep();
        if (rwsem_is_locked(&anon_vma->root->rwsem)) {
                anon_vma_lock_write(anon_vma);
                anon_vma_unlock_write(anon_vma);
        }

        kmem_cache_free(anon_vma_cachep, anon_vma);
}
```

Unassign anon_vma.

anon_vma Preparation

There are many places that ask you to prepare a anon_vma when preparing a free page for your users, and the main routines are as follows:

- Memory allocation in the event of a fault
    - do_cow_fault()
    - wp_page_copy()
    - do_anonymous_page()
- Stack Extensions
    - expand_upwards()
    - expand_downwards()
- Migration
    - migrate_vma_insert_page()

anon_vma_prepare()

include/linux/rmap.h

```
1  static inline int anon_vma_prepare(struct vm_area_struct *vma)
2  {
3          if (likely(vma->anon_vma))
4                  return 0;
5  
6          return __anon_vma_prepare(vma);
7  }
```

Prepare anon_vma in the VMA area.

- In line 3~4 of code, if the VMA already has a anon_vma ready and is in use, it returns success(0).
- In line 6 of code, prepare and attach anon_vma data structure to VMA to manage ANON pages.

__anon_vma_prepare()

mm/rmap.c

```
/**
 * __anon_vma_prepare - attach an anon_vma to a memory region
 * @vma: the memory region in question
 *
 * This makes sure the memory mapping described by 'vma' has
 * an 'anon_vma' attached to it, so that we can associate the
 * anonymous pages mapped into it with that anon_vma.
 *
 * The common case will be that we already have one, which
 * is handled inline by anon_vma_prepare(). But if
 * not we either need to find an adjacent mapping that we
 * can re-use the anon_vma from (very common when the only
 * reason for splitting a vma has been mprotect()), or we
 * allocate a new one.
 *
 * Anon-vma allocations are very subtle, because we may have
 * optimistically looked up an anon_vma in page_lock_anon_vma_read()
 * and that may actually touch the rwsem even in the newly
 * allocated vma (it depends on RCU to make sure that the
 * anon_vma isn't actually destroyed).
 *
 * As a result, we need to do proper anon_vma locking even
 * for the new allocation. At the same time, we do not want
 * to do any locking for the common case of already having
 * an anon_vma.
 *
 * This must be called with the mmap_lock held for reading.
 */
```

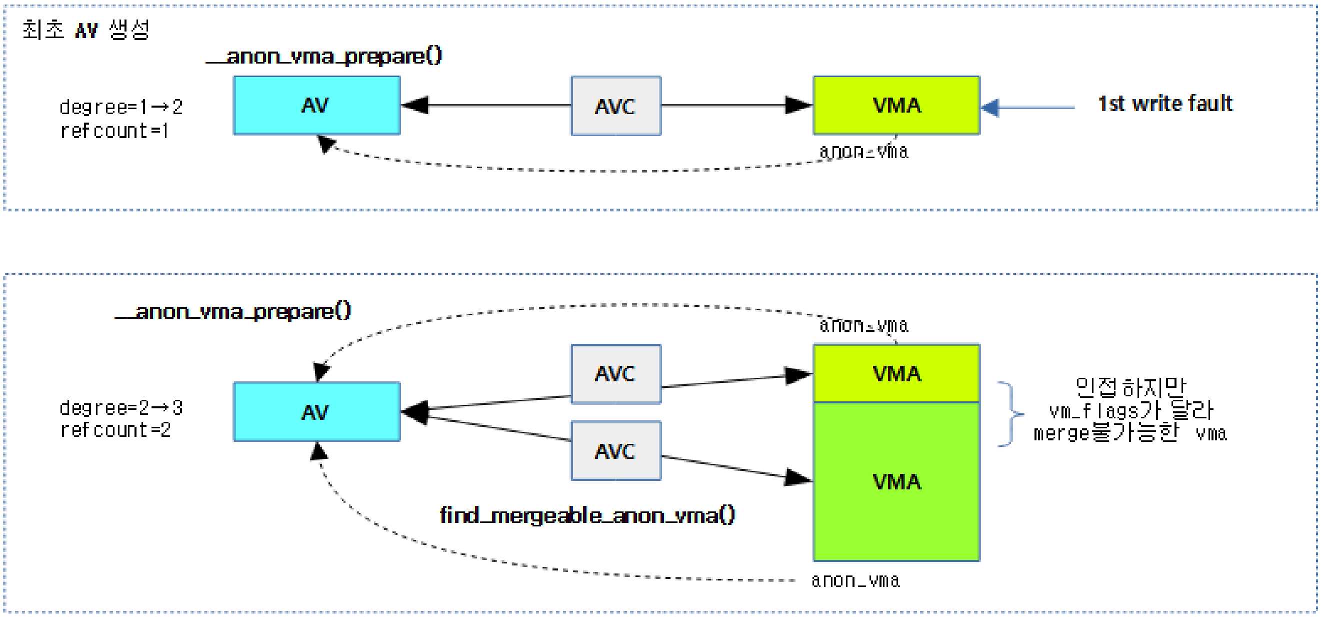
```
01 int __anon_vma_prepare(struct vm_area_struct *vma)
02 {
03         struct mm_struct *mm = vma->vm_mm;
04         struct anon_vma *anon_vma, *allocated;
05         struct anon_vma_chain *avc;
06 
07         might_sleep();
08 
09         avc = anon_vma_chain_alloc(GFP_KERNEL);
10         if (!avc)
11                 goto out_enomem;
12 
13         anon_vma = find_mergeable_anon_vma(vma);
14         allocated = NULL;
15         if (!anon_vma) {
16                 anon_vma = anon_vma_alloc();
17                 if (unlikely(!anon_vma))
18                         goto out_enomem_free_avc;
19                 allocated = anon_vma;
20         }
21 
22         anon_vma_lock_write(anon_vma);
23         /* page_table_lock to protect against threads */
24         spin_lock(&mm->page_table_lock);
25         if (likely(!vma->anon_vma)) {
26                 vma->anon_vma = anon_vma;
27                 anon_vma_chain_link(vma, avc, anon_vma);
28                 /* vma reference or self-parent link for new root */
29                 anon_vma->degree++;
30                 allocated = NULL;
31                 avc = NULL;
32         }
33         spin_unlock(&mm->page_table_lock);
34         anon_vma_unlock_write(anon_vma);
35 
36         if (unlikely(allocated))
37                 put_anon_vma(allocated);
38         if (unlikely(avc))
39                 anon_vma_chain_free(avc);
40 
41         return 0;
42 
43  out_enomem_free_avc:
44         anon_vma_chain_free(avc);
45  out_enomem:
46         return -ENOMEM;
47 }
```

Prepare and attach anon_vma data structure to manage ANON pages in VMA. The anon_vma is connected to the VMA via a anon_vma_chain.

- Assign anon_vma_chain in code lines 9~11.

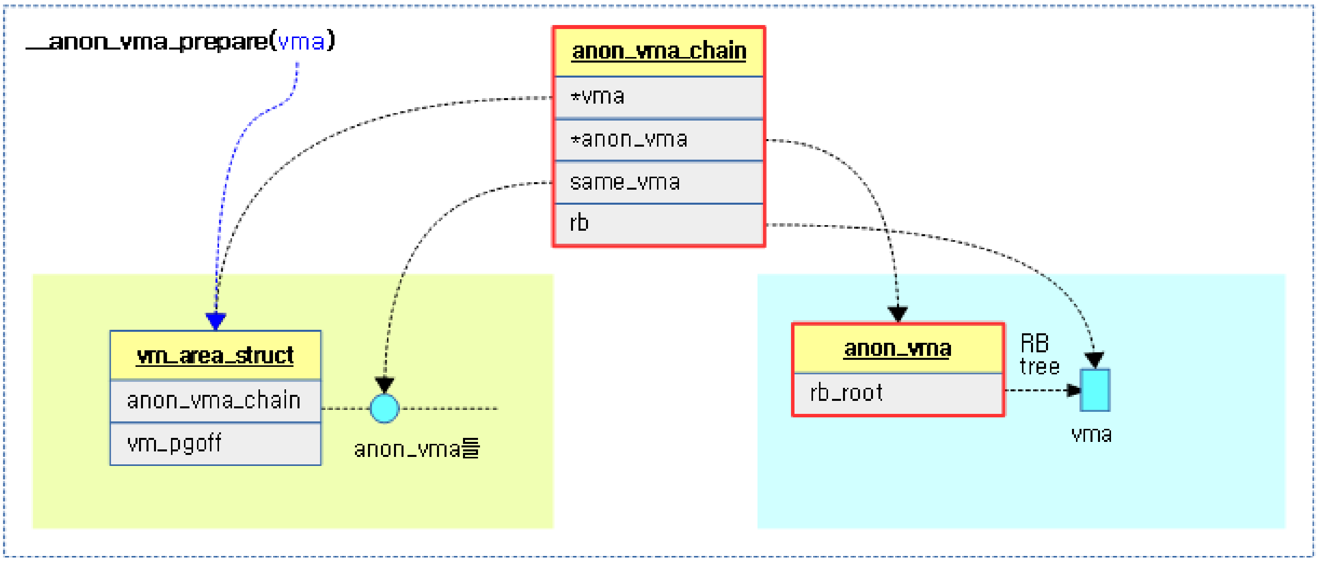
- In lines 13~20 of the code, find and fetch the mergable anon_vma of the neighboring VMA area, or assign a new anon_vma if it does not find.
- If you connect the anon_vma to the VMA for the first time after acquiring the lock in code lines 22~34, attach the anon_vma and anon_vma_chain to the VMA.
- If the anon_vma and anon_vma_chain allocated in line 36~39 of code fail to be attached to the VMA, the assigned anon_vma and anon_vma_chain are deallocated.

The following diagram shows the process of preparing an AV (anon_vma) by creating a new or using an existing AV.



(http://jake.dothome.co.kr/wp-content/uploads/2019/09/anon_vma_prepare-2a.png)

The following figure shows the process of assigning AV (anon_vma) to the first anon_vma to connect via VMA (vm_area_struct) and AVC (anon_vma_chain).



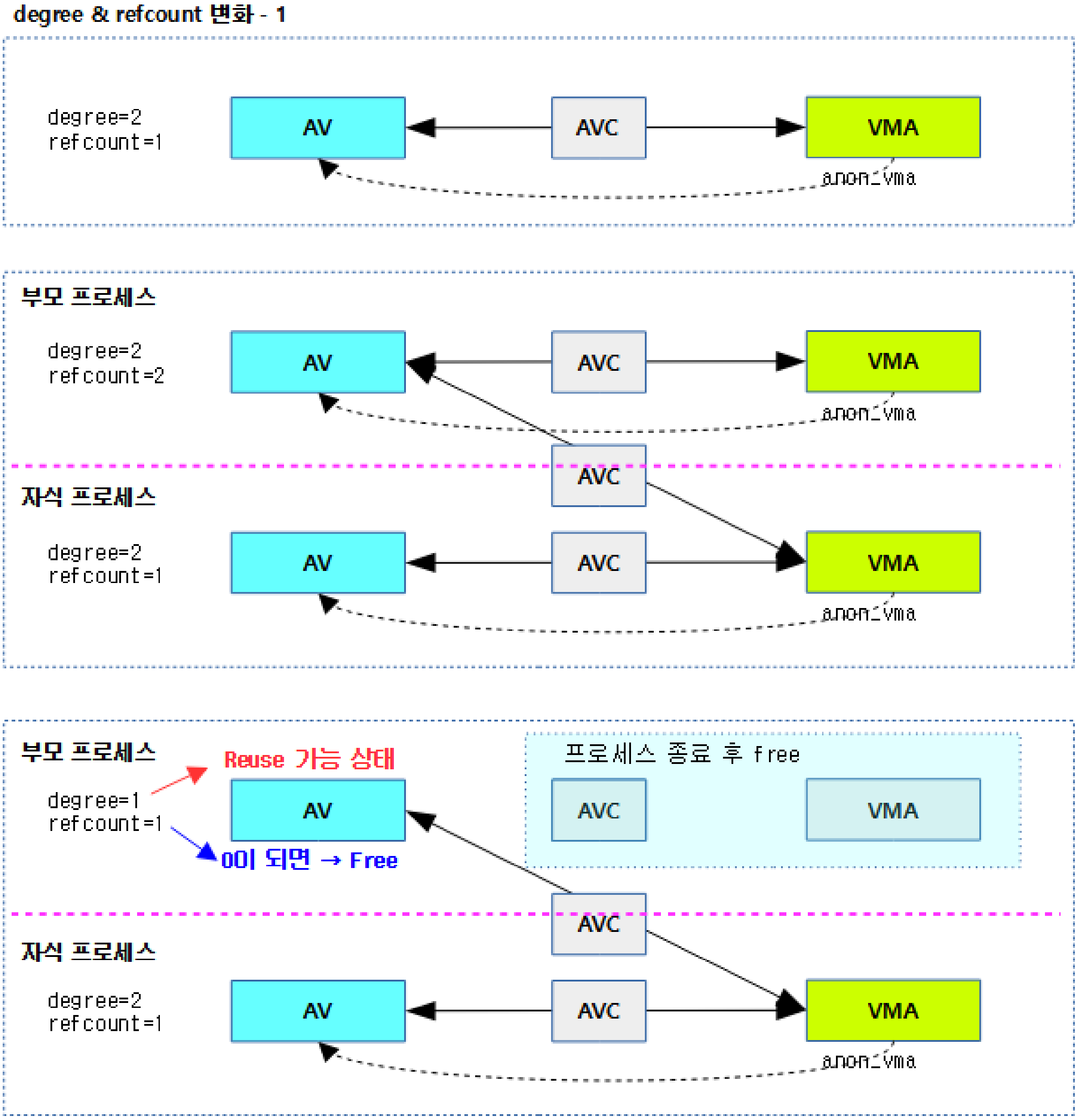
(http://jake.dothome.co.kr/wp-content/uploads/2019/09/anon_vma_prepare-1.png)

Managing the degree & refcount of AV (anon_vma)

Lifetime Management of AV (anon_vma)

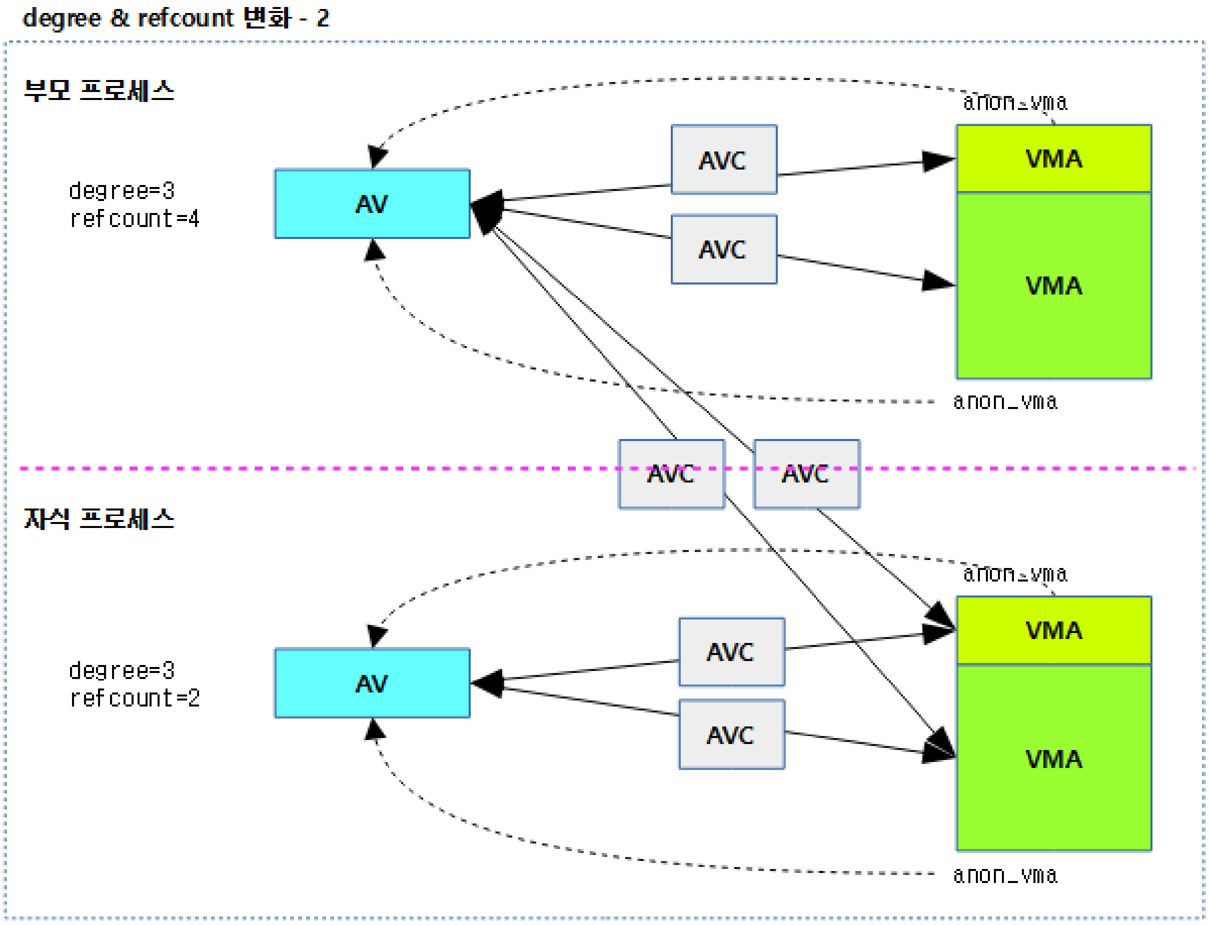
- refcount
  - When the reference counter reaches 0, it is destroyed.
  - It increases each time the VMA is connected via AVC (anon_vma_chain).
- degree
  - Each time you are designated as the owner of a VMA (an AV designated as vma->anon_vma), the degree increases.
  - When the AV is first created, it is 1, but it starts with 2 because it is immediately connected to the VMA and is designated as the owner of the VMA.
  - If this value is 1, the AV can be reused.
    - This is the case when only one forked child process is running and the parent process has terminated.
    - It has no vma and only one anon_vma child

The following figure shows the change in the degree and refcount of AV.



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

The following figure shows the change in degree and refcount of merged AV.



(http://jake.dothome.co.kr/wp-content/uploads/2019/09/anon_vma-2.png)

find_mergeable_anon_vma()

mm/mmap.c

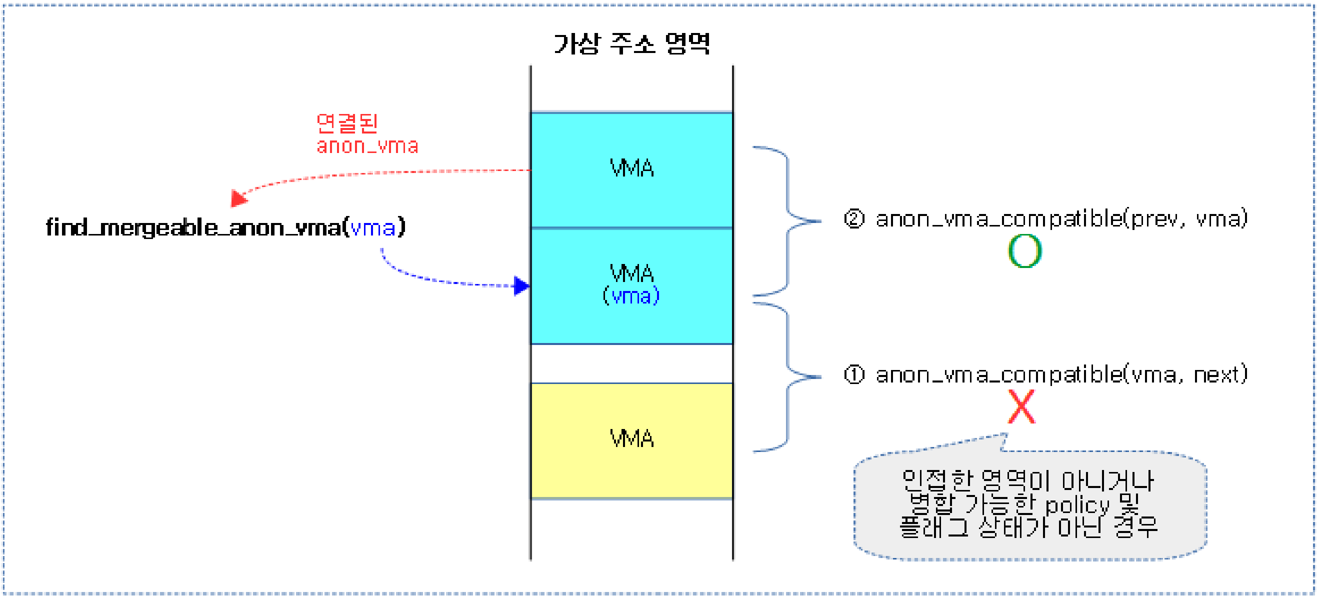
```
/*
 * find_mergeable_anon_vma is used by anon_vma_prepare, to check
 * neighbouring vmas for a suitable anon_vma, before it goes off
 * to allocate a new anon_vma.  It checks because a repetitive
 * sequence of mprotects and faults may otherwise lead to distinct
 * anon_vmas being allocated, preventing vma merge in subsequent
 * mprotect.
 */
```

```
struct anon_vma *find_mergeable_anon_vma(struct vm_area_struct *vma)
{
        struct anon_vma *anon_vma = NULL;

        /* Try next first. */
        if (vma->vm_next) {
                anon_vma = reusable_anon_vma(vma->vm_next, vma, vma->vm_next);
                if (anon_vma)
                        return anon_vma;
        }

        /* Try prev next. */
        if (vma->vm_prev)
                anon_vma = reusable_anon_vma(vma->vm_prev, vma->vm_prev, vma);

        /*
         * We might reach here with anon_vma == NULL if we can't find
         * any reusable anon_vma.
19          * There's no absolute need to look only at touching neighbours:
20          * we could search further afield for "compatible" anon_vmas.
21          * But it would probably just be a waste of time searching,
22          * or lead to too many vmas hanging off the same anon_vma.
23          * We're trying to allow mprotect remerging later on,
24          * not trying to minimize memory used for anon_vmas.
25          */
26         return anon_vma;
27 }
```

If there are mergable anon_vma in neighboring VMAs, it returns the anon_vma. If there are no mergable anon_vma, it returns null. (If it returns null, it will be created and used.)

- In line 6~10 of the code, if you can use a anon_vma in the neighboring VMA zone after the @vma area, you will know the anon_vma that neighbor VMA uses.
- In code lines 13~14, if a anon_vma can be used in a neighboring VMA zone before the @vma area, it knows the anon_vma used by that neighboring VMA.
- Returns anon_vma at line 26 of code.

The following figure shows the VMA returning a mergable anon_vma for two VMas that are adjacent to it.



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

reusable_anon_vma()

mm/mmap.c

```
/*
 * Do some basic sanity checking to see if we can re-use the anon_vma
 * from 'old'. The 'a'/'b' vma's are in VM order - one of them will be
 * the same as 'old', the other will be the new one that is trying
 * to share the anon_vma.
 *
 * NOTE! This runs with mm_sem held for reading, so it is possible that
 * the anon_vma of 'old' is concurrently in the process of being set up
 * by another page fault trying to merge _that_. But that's ok: if it
 * is being set up, that automatically means that it will be a singleton
 * acceptable for merging, so we can do all of this optimistically. But
 * we do that READ_ONCE() to make sure that we never re-load the pointer.
 *
 * IOW: that the "list_is_singular()" test on the anon_vma_chain only
 * matters for the 'stable anon_vma' case (ie the thing we want to avoid
 * is to return an anon_vma that is "complex" due to having gone through
 * a fork).
 *
 * We also make sure that the two vma's are compatible (adjacent,
 * and with the same memory policies). That's all stable, even with just
 * a read lock on the mm_sem.
 */
```

```
01 static struct anon_vma *reusable_anon_vma(struct vm_area_struct *old, struct vm_area_struct *a, struu
02 ct vm_area_struct *b)
03 {
04         if (anon_vma_compatible(a, b)) {
05                 struct anon_vma *anon_vma = READ_ONCE(old->anon_vma);
06 
07                 if (anon_vma && list_is_singular(&old->anon_vma_chain))
08                         return anon_vma;
09         }
10         return NULL;
11 }
```

If the anon_vma of the @a anon region and the @b anon region are mergable, it finds the anon_vma of the @old region and returns it.

- In line 4 of code, @a anon area and @b anon area are mergable.
- In line 5~8 of the code, if the anon_vma of the @old region exists, and only one AVC is registered in the @old region, it returns anon_vma.

anon_vma_compatible()

mm/mmap.c

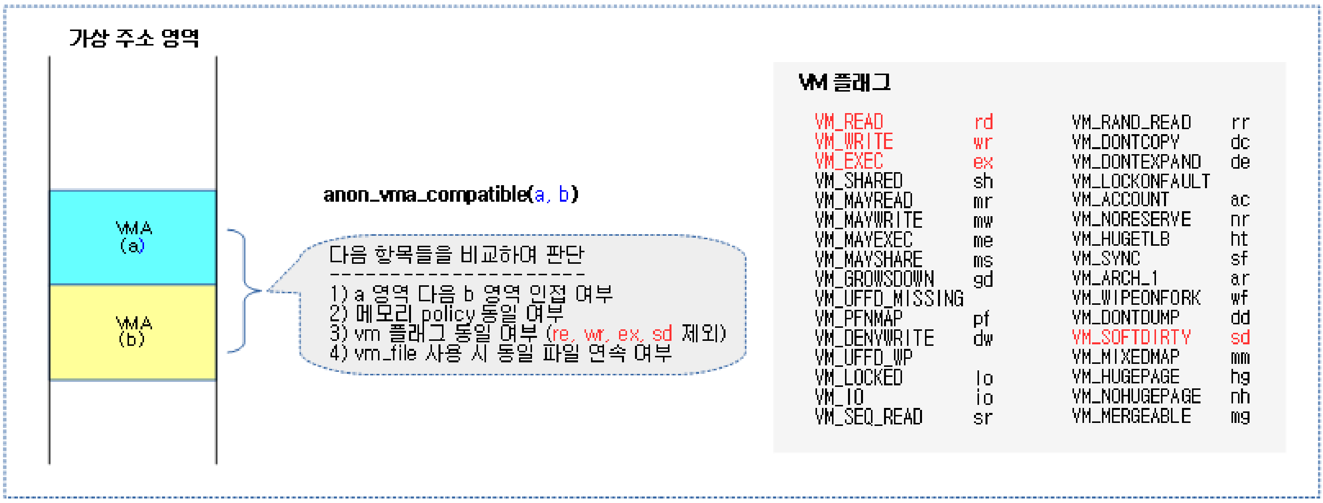
```
/*
 * Rough compatibility check to quickly see if it's even worth looking
 * at sharing an anon_vma.
 *
 * They need to have the same vm_file, and the flags can only differ
 * in things that mprotect may change.
 *
 * NOTE! The fact that we share an anon_vma doesn't _have_ to mean that
 * we can merge the two vma's. For example, we refuse to merge a vma if
 * there is a vm_ops->close() function, because that indicates that the
 * driver is doing some kind of reference counting. But that doesn't
 * really matter for the anon_vma sharing case.
 */
```

```
static int anon_vma_compatible(struct vm_area_struct *a, struct vm_area_struct *b)
{
        return a->vm_end == b->vm_start &&
                mpol_equal(vma_policy(a), vma_policy(b)) &&
                a->vm_file == b->vm_file &&
                !((a->vm_flags ^ b->vm_flags) & ~(VM_ACCESS_FLAGS | VM_SOFTDIRTY)) &&
                b->vm_pgoff == a->vm_pgoff + ((b->vm_start - a->vm_start) >> PAGE_SHIFT);
}
```

Returns whether the two anon zones can then be merged together or not. Returns Mergeable (1) if all of the following conditions are allowed:

- The two zones are attached to zone A and then area B.
- The VMA policy in both areas is the same.
- The use of flags other than read, write, exec, and softdirty in both areas is the same.
  - #define VM_ACCESS_FLAGS (VM_READ | VM_WRITE | VM_EXEC)
- The vm_file of the two domains is the same, and the vm_pgoff is assigned in the order of areas A and B.

The following shows the process of determining whether the two VMA zones of the virtual address zone can be used together.



다음 항목들을 비교하여 판단

1) a 영역 다음 b 영역 인접 여부
2) 메모리 policy 동일 여부
3) vm 플래그 동일 여부 (re, wr, ex, sd 제외)
4) vm_file 사용 시 동일 파일 연속 여부

VM_READ	rd	VM_RAND_READ	rr
VM_WRITE	wr	VM_DONTCOPY	dc
VM_EXEC	ex	VM_DONTEXPAND	de
VM_SHARED	sh	VM_LOCKONFAULT	
VM_MAYREAD	mr	VM_ACCOUNT	ac
VM_MAYWRITE	mw	VM_NORESERVE	nr
VM_MAYEXEC	me	VM_HUGETLB	ht
VM_MAYSHARE	ms	VM_SYNC	sf
VM_GROWSDOWN	gd	VM_ARCH_1	ar
VM_UFFD_MISSING		VM_WIPEONFORK	wf
VM_PFNMAP	pf	VM_DONTDUMP	dd
VM_DENYWRITE	dw	VM_SOFTDIRTY	sd
VM_UFFD_WP		VM_MIXEDMAP	mm
VM_LOCKED	lo	VM_HUGEPAGE	hg
VM_IO	io	VM_NOHUGEPAGE	nh
VM_SEQ_READ	sr	VM_MERGEABLE	mg

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

Links with anon_vma_chain

anon_vma_chain_alloc()

mm/rmap.c

```
static inline struct anon_vma_chain *anon_vma_chain_alloc(gfp_t gfp)
{
        return kmem_cache_alloc(anon_vma_chain_cachep, gfp);
}
```

Assign the prepared anon_vma_chain structure to the KMEM cache.

anon_vma_chain_free()

mm/rmap.c

```
static void anon_vma_chain_free(struct anon_vma_chain *anon_vma_chain)
{
        kmem_cache_free(anon_vma_chain_cachep, anon_vma_chain);
}
```

anon_vma_chain deallocates the struct and returns it to the KMEM cache.

anon_vma_chain_link()

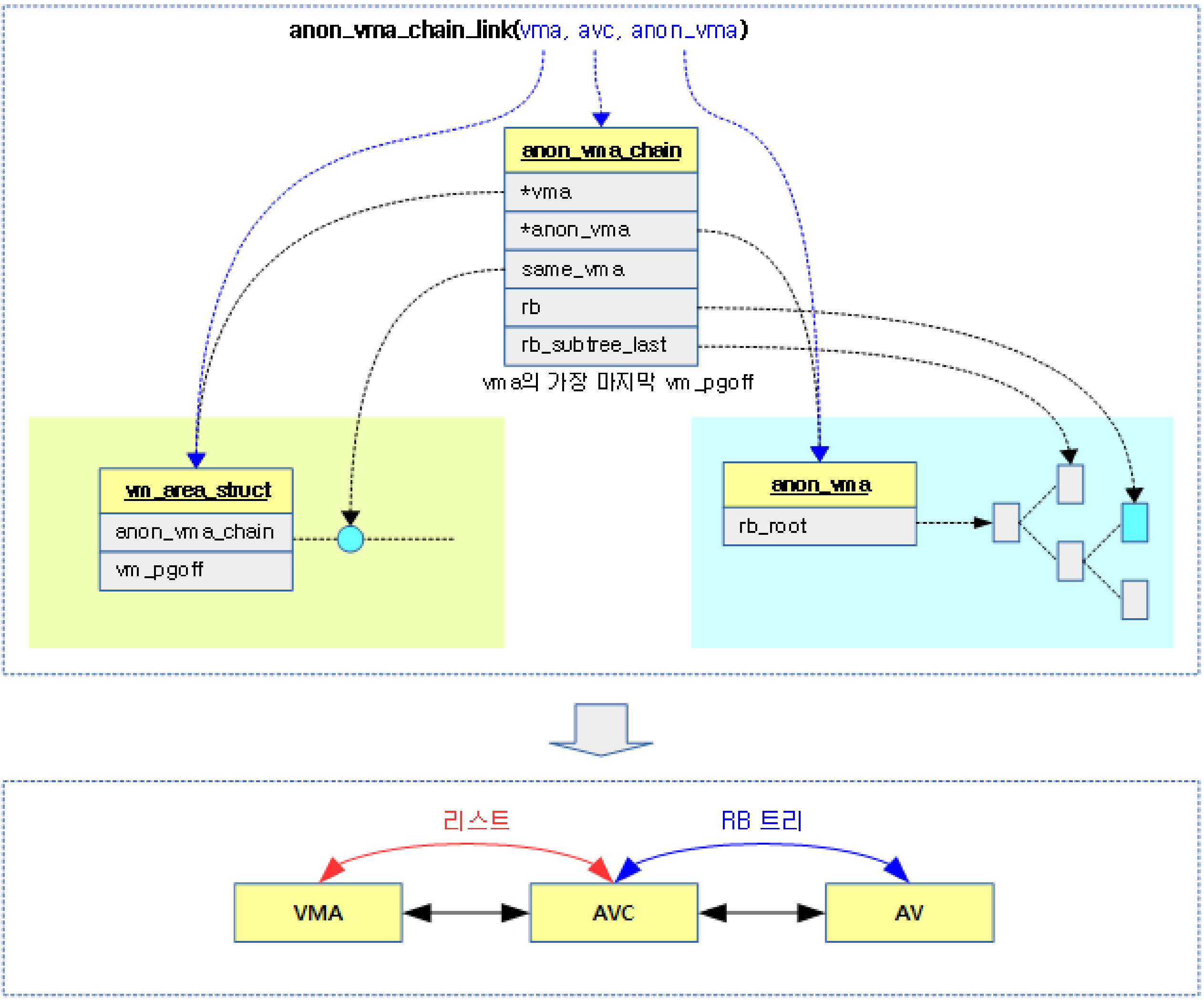
mm/rmap.c

```
1 static void anon_vma_chain_link(struct vm_area_struct *vma,
2                                 struct anon_vma_chain *avc,
3                                 struct anon_vma *anon_vma)
4 {
5         avc->vma = vma;
6         avc->anon_vma = anon_vma;
7         list_add(&avc->same_vma, &vma->anon_vma_chain);
8         anon_vma_interval_tree_insert(avc, &anon_vma->rb_root);
9 }
```

Link between VMA and anon_vma using AVC.

- In lines 5~6 of the code, specify that the @avc can point to @vma and @anon_vma.
- In line 7 of code, add the @avc to the anon_vma_chain list in the VMA.
- In line 8 of code, add the @avc to the RB tree in @anon_vma.

The following illustration shows VMAs and anon_vma being linked via AVC.



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

Add a mapping to an Anon page

page_add_anon_rmap()

mm/rmap.c

```
/**
 * page_add_anon_rmap - add pte mapping to an anonymous page
 * @page:       the page to add the mapping to
 * @vma:        the vm area in which the mapping is added
 * @address:    the user virtual address mapped
 * @compound:   charge the page as compound or small page
 *
 * The caller needs to hold the pte lock, and the page must be locked in
 * the anon_vma case: to serialize mapping,index checking after setting,
 * and to ensure that PageAnon is not being upgraded racily to PageKsm
 * (but PageKsm is never downgraded to PageAnon).
 */
```

```
void page_add_anon_rmap(struct page *page,
        struct vm_area_struct *vma, unsigned long address, bool compound)
{
        do_page_add_anon_rmap(page, vma, address, compound ? RMAP_COMPOUND : 0);
}
```

Add a reverse mapping to a shared anon page.

do_page_add_anon_rmap()

Note: This function is only used to add a reverse mapping from do_swap_page() to a non-shared anon page. In other cases, you should use the above page_add_anon_rmap() function.

mm/rmap.c

```
/*
 * Special version of the above for do_swap_page, which often runs
 * into pages that are exclusively owned by the current process.
 * Everybody else should continue to use page_add_anon_rmap above.
 */
```

```
void do_page_add_anon_rmap(struct page *page,
        struct vm_area_struct *vma, unsigned long address, int flags)
{
        bool compound = flags & RMAP_COMPOUND;
        bool first;

        if (unlikely(PageKsm(page)))
                lock_page_memcg(page);
        else
                VM_BUG_ON_PAGE(!PageLocked(page), page);

        if (compound) {
                atomic_t *mapcount;
                VM_BUG_ON_PAGE(!PageLocked(page), page);
                VM_BUG_ON_PAGE(!PageTransHuge(page), page);
                mapcount = compound_mapcount_ptr(page);
                first = atomic_inc_and_test(mapcount);
        } else {
                first = atomic_inc_and_test(&page->_mapcount);
        }

        if (first) {
                int nr = compound ? thp_nr_pages(page) : 1;
                /*
                 * We use the irq-unsafe __{inc|mod}_zone_page_stat because
                 * these counters are not modified in interrupt context, and
                 * pte lock(a spinlock) is held, which implies preemption
                 * disabled.
                 */
                if (compound)
                        __inc_node_page_state(page, NR_ANON_THPS);
                __mod_node_page_state(page_pgdat(page), NR_ANON_MAPPED, nr);
        }
        if (unlikely(PageKsm(page))) {
                unlock_page_memcg(page);
                return;
        }

        /* address might be in next vma when migration races vma_adjust */
        if (first)
                __page_set_anon_rmap(page, vma, address,
                                flags & RMAP_EXCLUSIVE);
        else
                __page_check_anon_rmap(page, vma, address);
}
```

Add a reverse mapping to the anon page.

- In line 4 of code, check whether the flag has a compound request.
- In line 7~10 of the code, for the KSM page, lock the page and memcg binding.
- In lines 12~20 of the code, increment the mapping count for that page by 1 and find out whether it is mapped the first time.
- On lines 22~33 of the code, increment the nr_anon_mapped counter by the number of pages in the first mapping. If it is a compound page, the nr_anon_thps counter is also incremented by 1.
- In the case of the KSM page in line 34~37 of code, unlock the function from above and exit the function.
- If this is the first mapping in line 40~44 of the code, map the page to the new anonymous rmap. Otherwise, it does nothing but debug.

page_add_new_anon_rmap()

mm/rmap.c

```
/**
 * page_add_new_anon_rmap - add pte mapping to a new anonymous page
 * @page:       the page to add the mapping to
 * @vma:        the vm area in which the mapping is added
 * @address:    the user virtual address mapped
 * @compound:   charge the page as compound or small page
 *
 * Same as page_add_anon_rmap but must only be called on *new* pages.
 * This means the inc-and-test can be bypassed.
 * Page does not have to be locked.
 */
```

```
01 void page_add_new_anon_rmap(struct page *page,
02         struct vm_area_struct *vma, unsigned long address, bool compound)
03 {
04         int nr = compound ? thp_nr_pages(page) : 1;
05 
06         VM_BUG_ON_VMA(address < vma->vm_start || address >= vma->vm_end, vma);
07         __SetPageSwapBacked(page);
08         if (compound) {
09                 VM_BUG_ON_PAGE(!PageTransHuge(page), page);
10                 /* increment count (starts at -1) */
11                 atomic_set(compound_mapcount_ptr(page), 0);
12                 if (hpage_pincount_available(page))
13                         atomic_set(compound_pincount_ptr(page), 0);
14 
15                 __mod_lruvec_page_state(page, NR_ANON_THPS, nr);
16         } else {
17                 /* Anon THP always mapped first with PMD */
18                 VM_BUG_ON_PAGE(PageTransCompound(page), page);
19                 /* increment count (starts at -1) */
20                 atomic_set(&page->_mapcount, 0);
21         }
22         __mod_lruvec_page_state(page_pgdat(page), NR_ANON_MAPPED, nr);
23         __page_set_anon_rmap(page, vma, address, 1);
24 }
```

Assign rmap to the task-specific anon page.

- In line 7 of the code, add the PG_swapbacked flag because this is the newly assigned anon page.

- In line 8~15 of the code, if it is a compound page, reset the compound mapping counter to 0 and increment the NR_ANON_THPS counter.
- In code lines 16~21, reset the mapping counter to 0 if it is not a compound page.
- In lines 22~23 of the code, increment the NR_ANON_MAPPED counter and set anon rmap on the page.
    - page->mapping = anon_vma
    - page->index = linear_page_index(vma, address)

__page_set_anon_rmap()

mm/rmap.c

```
/**
 * __page_set_anon_rmap - set up new anonymous rmap
 * @page:       Page or Hugepage to add to rmap
 * @vma:        VM area to add page to.
 * @address:    User virtual address of the mapping
 * @exclusive:  the page is exclusively owned by the current process
 */
```

```
01 static void __page_set_anon_rmap(struct page *page,
02         struct vm_area_struct *vma, unsigned long address, int exclusive)
03 {
04         struct anon_vma *anon_vma = vma->anon_vma;
05 
06         BUG_ON(!anon_vma);
07 
08         if (PageAnon(page))
09                 return;
10 
11         /*
12          * If the page isn't exclusively mapped into this vma,
13          * we must use the _oldest_ possible anon_vma for the
14          * page mapping!
15          */
16         if (!exclusive)
17                 anon_vma = anon_vma->root;
18 
19         /*
20          * page_idle does a lockless/optimistic rmap scan on page->mapping.
21          * Make sure the compiler doesn't split the stores of anon_vma and
22          * the PAGE_MAPPING_ANON type identifier, otherwise the rmap code
23          * could mistake the mapping for a struct address_space and crash.
24         /*
25         anon_vma = (void *) anon_vma + PAGE_MAPPING_ANON;
26         WRITE_ONCE(page->mapping = (struct address_space *) anon_vma);
27         page->index = linear_page_index(vma, address);
28 }
```

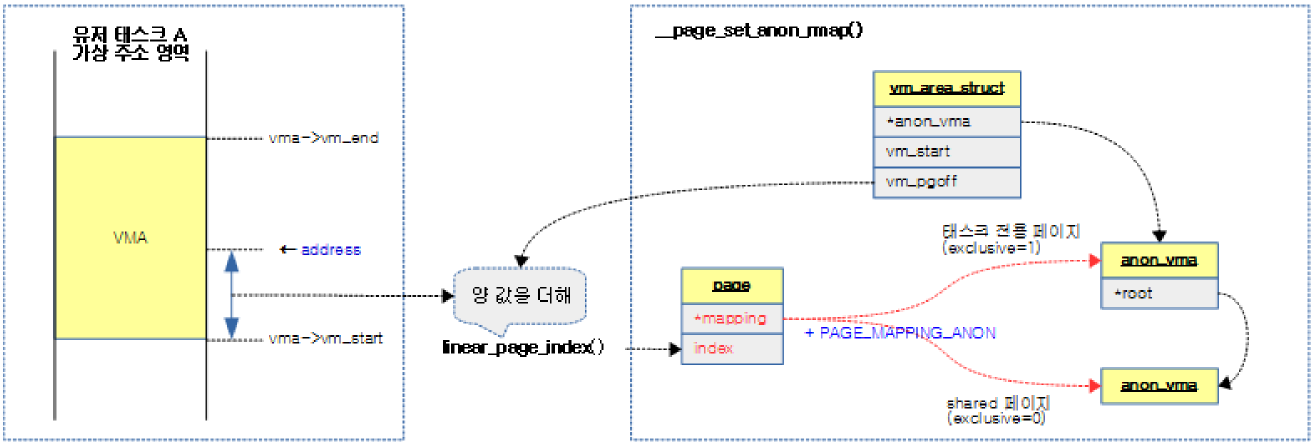
Specify the page as anonymous rmap.

- In line 8~9 of the code, if the page is already anon mapped, exit the function.
- In lines 16~17 of the code, if the page is a shared page and not dedicated to the current task, use the root anon_vma.
- In lines 25~27 of the code, map the anon to the page.

- page->mapping is specified by adding a PAGE_PAPPING_ANON flag to the anon_vma pointer.
- page->index specifies a value for the page offset + (address pfn – vm start pfn) specified in the VMA.

The following illustration shows the process of mapping an anon page to an rmap.

- Pages mapped to anon return true via the PageAnon(page) function.



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

linear_page_index()

include/linux/pagemap.h

```
static inline pgoff_t linear_page_index(struct vm_area_struct *vma,
                                        unsigned long address)
{
        pgoff_t pgoff;
        if (unlikely(is_vm_hugetlb_page(vma)))
                return linear_hugepage_index(vma, address);
        pgoff = (address - vma->vm_start) >> PAGE_SHIFT;
        pgoff += vma->vm_pgoff;
        return pgoff;
}
```

Returns the index value of the page offset corresponding to the requested address in the VMA area plus the VMA->vm_pgoff.

Remove an RMAP mapping from a page

page_remove_rmap()

mm/rmap.c

```
/**
 * page_remove_rmap - take down pte mapping from a page
 * @page:       page to remove mapping from
 * @compound:   uncharge the page as compound or small page
 *
 * The caller needs to hold the pte lock.
 */
```

```
void page_remove_rmap(struct page *page, bool compound)
{
        lock_page_memcg(page);

        if (!PageAnon(page)) {
                page_remove_file_rmap(page, compound);
                goto out;
        }

        if (compound) {
                page_remove_anon_compound_rmap(page);
                goto out;
        }

        /* page still mapped by someone else? */
        if (!atomic_add_negative(-1, &page->_mapcount))
                goto out;

        /*
         * We use the irq-unsafe __{inc|mod}_zone_page_stat because
         * these counters are not modified in interrupt context, and
         * pte lock(a spinlock) is held, which implies preemption disabled.
         */
        __dec_lruvec_page_state(page, NR_ANON_MAPPED);

        if (unlikely(PageMlocked(page)))
                clear_page_mlock(page);

        if (PageTransCompound(page))
                deferred_split_huge_page(compound_head(page));

        /*
         * It would be tidy to reset the PageAnon mapping here,
         * but that might overwrite a racing page_add_anon_rmap
         * which increments mapcount after us but sets mapping
         * before us: so leave the reset to free_unref_page,
         * and remember that it's only reliable while mapped.
         * Leaving it set also helps swapoff to reinstate ptes
         * faster for those pages still in swapcache.
         */
out:
        unlock_page_memcg(page);
}
```

Remove the reverse mapping (rmap) from the page.

page_remove_file_rmap()

mm/rmap.c

```
static void page_remove_file_rmap(struct page *page, bool compound)
{
        int i, nr = 1;

        VM_BUG_ON_PAGE(compound && !PageHead(page), page);
        lock_page_memcg(page);

        /* Hugepages are not counted in NR_FILE_MAPPED for now. */
        if (unlikely(PageHuge(page))) {
                /* hugetlb pages are always mapped with pmds */
                atomic_dec(compound_mapcount_ptr(page));
                return;
        }

        /* page still mapped by someone else? */
        if (compound && PageTransHuge(page)) {
                int nr_pages = thp_nr_pages(page);

                for (i = 0, nr = 0; i < HPAGE_PMD_NR; i++) {
                        if (atomic_add_negative(-1, &page[i]._mapcount))
                                nr++;
                }
                if (!atomic_add_negative(-1, compound_mapcount_ptr(page)))
                        return;
                if (PageSwapBacked(page))
                        __mod_lruvec_page_state(page, NR_SHMEM_PMDMAPPED,
                                                -nr_pages);
                else
                        __mod_lruvec_page_state(page, NR_FILE_PMDMAPPED,
                                                -nr_pages);
        } else {
                if (!atomic_add_negative(-1, &page->_mapcount))
                        return;
        }

        /*
         * We use the irq-unsafe __{inc|mod}_lruvec_page_state because
         * these counters are not modified in interrupt context, and
         * pte lock(a spinlock) is held, which implies preemption disabled.
         */
        __mod_lruvec_page_state(page, NR_FILE_MAPPED, -nr);

        if (unlikely(PageMlocked(page)))
                clear_page_mlock(page);
out:
        unlock_page_memcg(page);
}
```

Remove the reverse mapping (rmap) of the file page.

Child process fork creates and connects anon_vma

anon_vma_fork()

mm/rmap.c

```
/*
 * Attach vma to its own anon_vma, as well as to the anon_vmas that
 * the corresponding VMA in the parent process is attached to.
 * Returns 0 on success, non-zero on failure.
 */
```

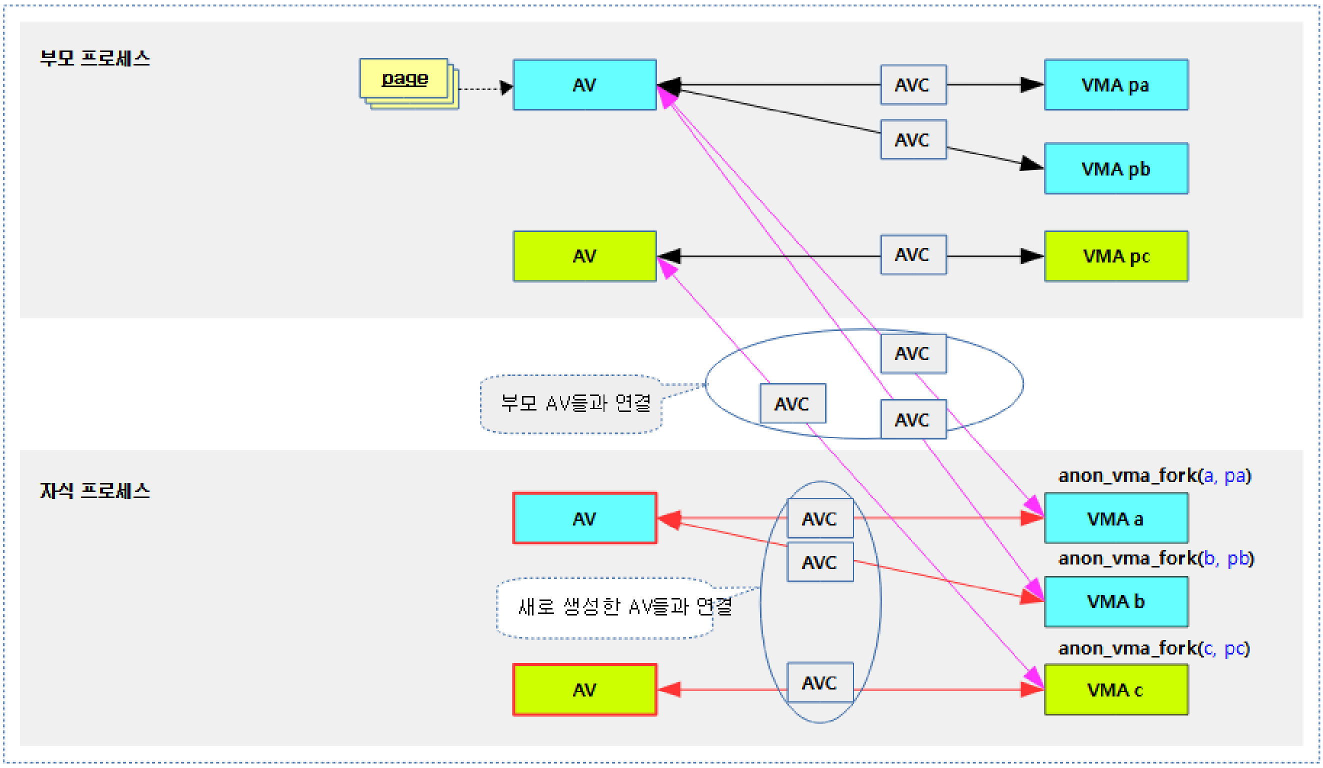
```
int anon_vma_fork(struct vm_area_struct *vma, struct vm_area_struct *pvma)
{
        struct anon_vma_chain *avc;
        struct anon_vma *anon_vma;
        int error;

        /* Don't bother if the parent process has no anon_vma here. */
        if (!pvma->anon_vma)
                return 0;

11          /* Drop inherited anon_vma, we'll reuse existing or allocate new. */
12          vma->anon_vma = NULL;
13
14          /*
15           * First, attach the new VMA to the parent VMA's anon_vmas,
16           * so rmap can find non-COWed pages in child processes.
17           */
18          error = anon_vma_clone(vma, pvma);
19          if (error)
20                  return error;
21
22          /* An existing anon_vma has been reused, all done then. */
23          if (vma->anon_vma)
24                  return 0;
25
26          /* Then add our own anon_vma. */
27          anon_vma = anon_vma_alloc();
28          if (!anon_vma)
29                  goto out_error;
30          avc = anon_vma_chain_alloc(GFP_KERNEL);
31          if (!avc)
32                  goto out_error_free_anon_vma;
33
34          /*
35           * The root anon_vma's rwsem is the lock actually used when we
36           * lock any of the anon_vmas in this anon_vma tree.
37           */
38          anon_vma->root = pvma->anon_vma->root;
39          anon_vma->parent = pvma->anon_vma;
40          /*
41           * With refcounts, an anon_vma can stay around longer than the
42           * process it belongs to. The root anon_vma needs to be pinned until
43           * this anon_vma is freed, because the lock lives in the root.
44           */
45          get_anon_vma(anon_vma->root);
46          /* Mark this anon_vma as the one where our new (COWed) pages go. */
47          vma->anon_vma = anon_vma;
48          anon_vma_lock_write(anon_vma);
49          anon_vma_chain_link(vma, avc, anon_vma);
50          anon_vma->parent->degree++;
51          anon_vma_unlock_write(anon_vma);
52
53          return 0;
54
55   out_error_free_anon_vma:
56          put_anon_vma(anon_vma);
57   out_error:
58          unlink_anon_vmas(vma);
59          return -ENOMEM;
60  }
```

The forked @vma created in the child process is newly allocated to the number of anon_vma existing in the parent @pvma, and then linked. In addition, anon_vma corresponding to @vma are reused or allocated.

- In line 8~9 of the code, if the parent VMA has never specified a anon_vma, it returns success(0).
- In line 12 of code, we put null in order to reset the anon_vma of the VMA that we created by inheriting from the parent task.
- In lines 18~20 of code, link the anon_vma linked to the parent VMAs and the newly clone VMAs.
- If the anon_vma is reused in lines 23~24 of the code, returns success (0).
- In lines 27~32 of the code, we assign the anon_vma and anon_vma_chain structs.

- In lines 38~39 of the code, specify the root of the anon_vma and the parent.
- In lines 45~53 of code, link the anon_vma to the vm_area_struct via anon_vma_chain and return success(0).

The following figure shows the result of anon_vma_fork() being called three times as the child process is forked.



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

anon_vma_clone()

The main call pass for this function is as follows:

- The task is forked to clone the VMA of the parent task.
  - anon_vma_fork()
- When splitting a VMA, clone an existing VMA
  - __split_vma()

mm/rmap.c

```
01 /*
02  * Attach the anon_vmas from src to dst.
03  * Returns 0 on success, -ENOMEM on failure.
04  *
05  * anon_vma_clone() is called by __vma_adjust(), __split_vma(), copy_vma() and
06  * anon_vma_fork(). The first three want an exact copy of src, while the last
07  * one, anon_vma_fork(), may try to reuse an existing anon_vma to prevent
08  * endless growth of anon_vma. Since dst->anon_vma is set to NULL before call,
09  * we can identify this case by checking (!dst->anon_vma && src->anon_vma).
```

```
 *
 * If (!dst->anon_vma && src->anon_vma) is true, this function tries to find
 * and reuse existing anon_vma which has no vmas and only one child anon_vma.
 * This prevents degradation of anon_vma hierarchy to endless linear chain in
 * case of constantly forking task. On the other hand, an anon_vma with more
 * than one child isn't reused even if there was no alive vma, thus rmap
 * walker has a good chance of avoiding scanning the whole hierarchy when it
 * searches where page is mapped.
 */
```

```
int anon_vma_clone(struct vm_area_struct *dst, struct vm_area_struct *src)
{
        struct anon_vma_chain *avc, *pavc;
        struct anon_vma *root = NULL;

        list_for_each_entry_reverse(pavc, &src->anon_vma_chain, same_vma) {
                struct anon_vma *anon_vma;

                avc = anon_vma_chain_alloc(GFP_NOWAIT | __GFP_NOWARN);
                if (unlikely(!avc)) {
                        unlock_anon_vma_root(root);
                        root = NULL;
                        avc = anon_vma_chain_alloc(GFP_KERNEL);
                        if (!avc)
                                goto enomem_failure;
                }
                anon_vma = pavc->anon_vma;
                root = lock_anon_vma_root(root, anon_vma);
                anon_vma_chain_link(dst, avc, anon_vma);

                /*
                 * Reuse existing anon_vma if its degree lower than two,
                 * that means it has no vma and only one anon_vma child.
                 *
                 * Do not chose parent anon_vma, otherwise first child
                 * will always reuse it. Root anon_vma is never reused:
                 * it has self-parent reference and at least one child.
                 */
                if (!dst->anon_vma && src->anon_vma &&
                    anon_vma != src->anon_vma && anon_vma->degree < 2) dst->anon_vma = anon_vma;
        }
        if (dst->anon_vma)
                dst->anon_vma->degree++;
        unlock_anon_vma_root(root);
        return 0;

 enomem_failure:
        /*
         * dst->anon_vma is dropped here otherwise its degree can be incorrectly
         * decremented in unlink_anon_vmas().
         * We can safely do this because callers of anon_vma_clone() don't care
         * about dst->anon_vma if anon_vma_clone() failed.
         */
        dst->anon_vma = NULL;
        unlink_anon_vmas(dst);
        return -ENOMEM;
}
```

@src links @dst VMAs with anon_vma linked to the VMas.

- In line 6 of the code, @src traverss the PVCs as many anon_vma connected to the VMA.
    - @src tour the AVCs linked to the VMA's anon_vma_chain list.
- Assign anon_vma_chain in code lines 9~16. If the allocation fails, anon_vma unlock the root lock and try the allocation again.
- On line 17 of the code, we know the anon_vma connected to the PVC that is traversing.
- At line 18 of the code, lock root anon_vma.
- Link the clone @dst VMA in line 19 using the newly assigned anon_vma_chain between the VMa and the traversing anon_vma.
- This is a patch to prevent anon_vma high raki from continuing to increase endlessly in lines 29~31 of the code. anon_vma who do not own a VMA and have only one child anon_vma for anon_vma who are grandparents or more are allowed to reuse.
- If the anon_vma is specified in the @dst VMA in code lines 32~33, the degree is increased.

Patches to prevent endless growth of anon_vma from repeated fork hierarchy

Consultation:

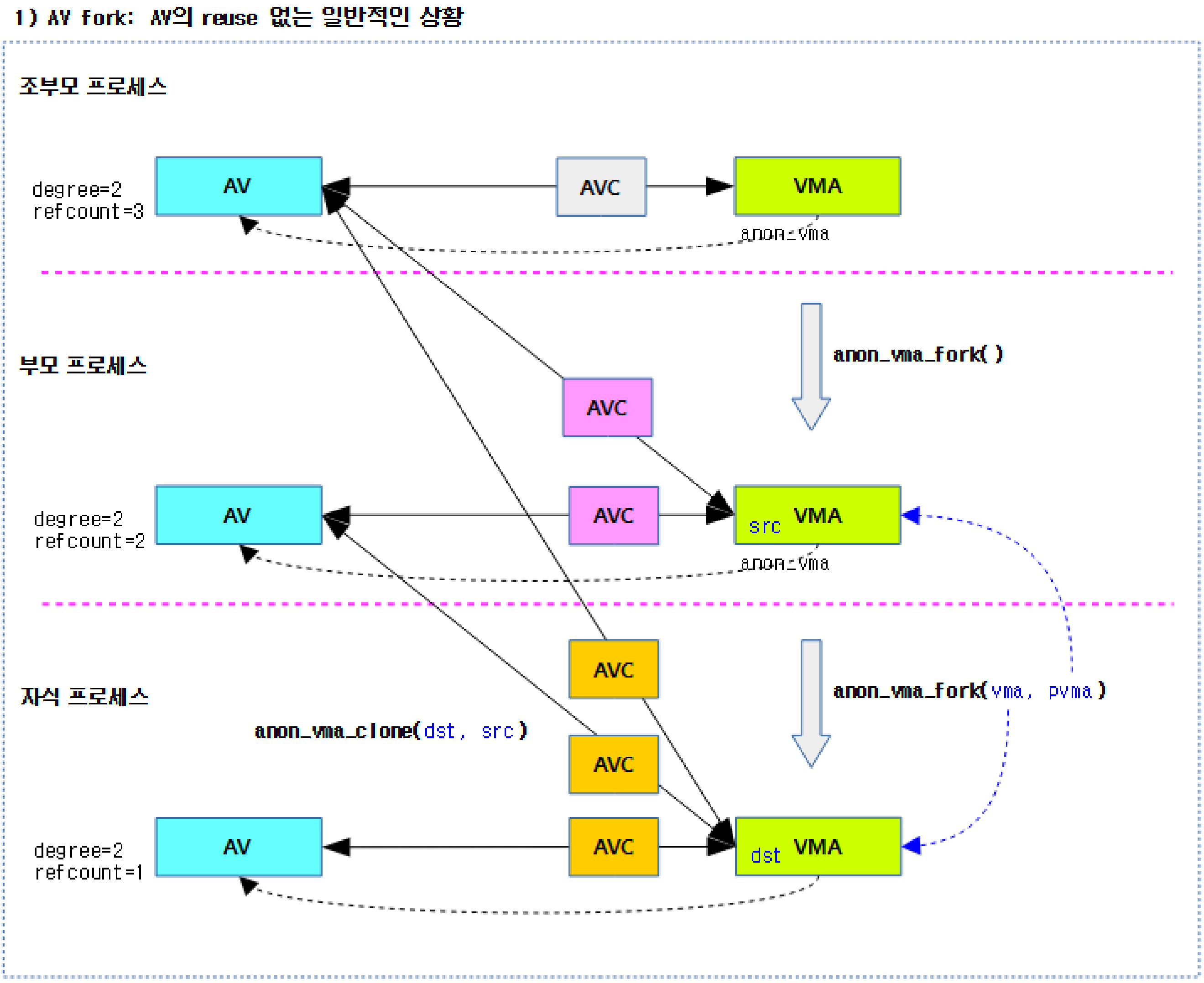
- mm: prevent endless growth of anon_vma hierarchy (https://github.com/torvalds/linux/commit/7a3ef208e662f4b63d43a23f61a64a129c525bbc) (2015, v3.19-rc4)
- mm/rmap.c: don't reuse anon_vma if we just want a copy (https://github.com/torvalds/linux/commit/47b390d23bf81894395c8773acf6f73c66465dc4) (2019, v5.5-rc1)
- mm/rmap.c: reuse mergeable anon_vma as parent when fork (https://github.com/torvalds/linux/commit/4e4a9eb921332b9d1edd99f76998f99f36b195f7) (2019, v5.5-rc1)
- Repeated fork() causes SLAB to grow without bound (https://lore.kernel.org/all/20120816024610.GA5350@evergreen.ssec.wisc.edu/T/#u)

The anon_vma was constantly increasing due to repeated forks as follows, and the anon_vma was patched by reuse.

```
#include <unistd.h>

int main(int argc, char *argv[])
{
        pid_t pid;

        while (1) {
                pid = fork();
                if (pid == -1) {
                        /* error */
                        return 1;
                }
                if (pid) {
                        /* parent */
                        sleep(2);
                        break;
                }
                else {
                        /* child */
                        sleep(1);
                }
        }

        return 0;
}
```

The following figure shows the process of connecting AV when forked from the grandparent process to the parent and child.

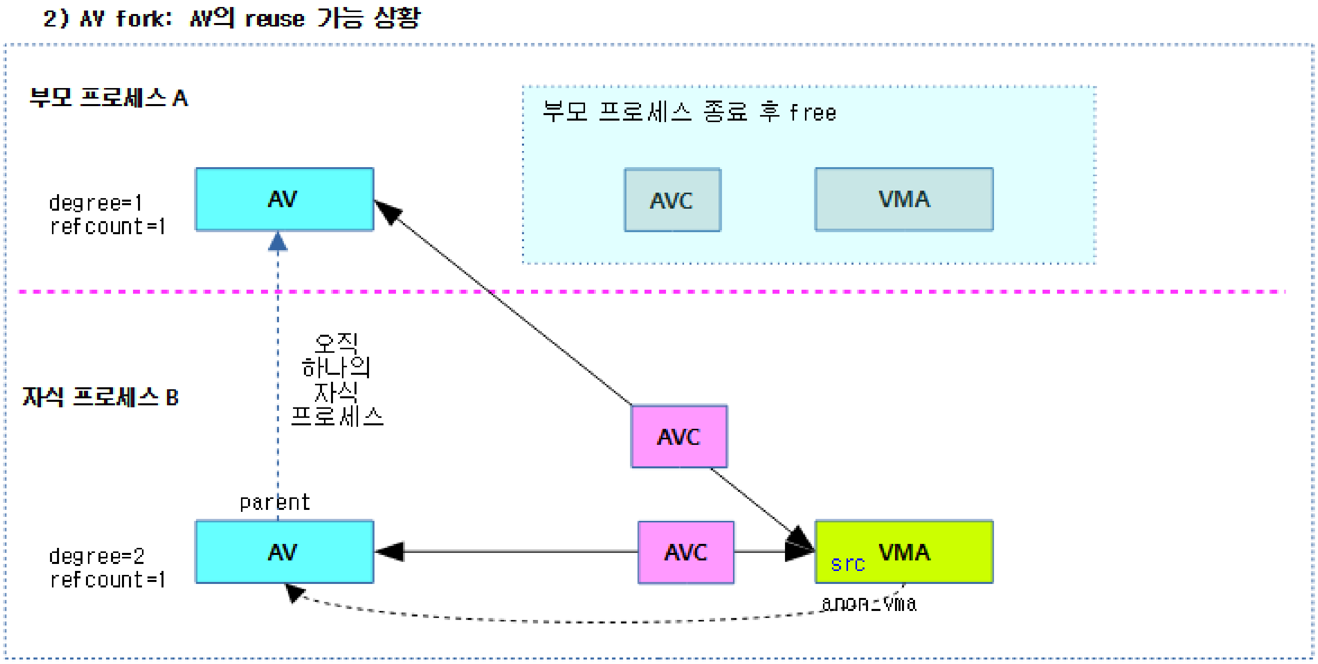
- Only one VMA and one AV were used for the grandparent process to demonstrate the fork process in detail.



(http://jake.dothome.co.kr/wp-content/uploads/2019/09/anon_vma_fork-3.png)

The following figure shows the state in which the AV can be reused after the parent process ends.

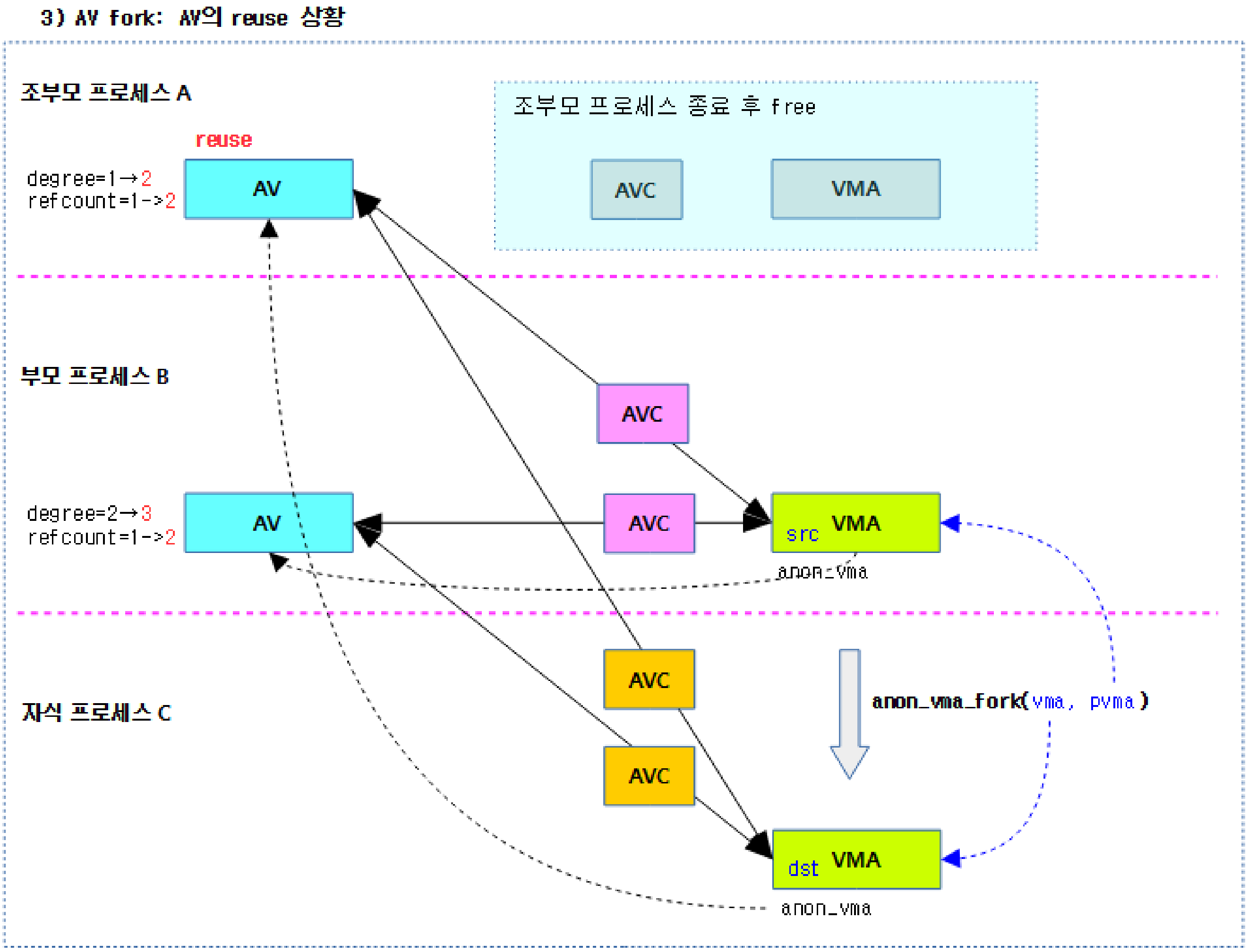
- If the forked child process B terminates, the parent process's AVs are free without reuse.



(http://jake.dothome.co.kr/wp-content/uploads/2019/09/anon_vma_fork-4.png)

The following figure shows how a child process reuses AVs from the grandparent process.

- Degrees of 1 must be for AVs used by grandparents or higher, excluding parent processes.



(http://jake.dothome.co.kr/wp-content/uploads/2019/09/anon_vma_fork-5.png)

consultation

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

- The case of the overly anonymous anon_vma (https://lwn.net/Articles/383162/) (2010) | LWN.net
- Virtual Memory II: the return of objrmap (https://lwn.net/Articles/75198/) (2004) | LWN.net
- Reverse mapping anonymous pages – again (https://lwn.net/Articles/77106/) (2004)| LWN.net
- Linux Memory Management | Columbia.edu – Download PDF (http://www.cs.columbia.edu/~krj/os/lectures/L17-LinuxPaging.pdf)

4 thoughts to “Rmap -1- (Reverse Mapping)”

SHY
2020-02-04 14:16 (http://jake.dothome.co.kr/rmap-1/#comment-231184)

Thanks for the wonderful information!!

RESPONSE (/RMAP-1/?REPLYTOCOM=231184#RESPOND)

MOON YOUNG-IL (HTTP://JAKE.DOTHOME.CO.KR)
2020-02-04 21:51 (http://jake.dothome.co.kr/rmap-1/#comment-231230)

I appreciate it. I wish you many accomplishments.

RESPONSE (/RMAP-1/?REPLYTOCOM=231230#RESPOND)

IPARAN (HTTPS://WWW.BHRAL.COM/)
2021-06-20 14:32 (http://jake.dothome.co.kr/rmap-1/#comment-305623)

Hello! Moon Young-il!

As for the compaction based on kernel version v16.5, which was a great help to me because I participated in the 1th study last time, the subsystem is finished until the review, and
I am looking hard at the rmap subsystem.

I remember that you corrected it last time I saw it, but I was wondering if it was not reflected, so I inquire in the comments.

At the beginning of the article

Rmap is a method of using a physical address and backmapping it to a virtual address.
Find the page table entry used by each VM for all tasks that use the shared anon page and use it to remove the mapping.

There is a kind of reverse mapping described in the very next paragraph,

In the very first line, you told me that you might
mistake it for reverse mapping in the case of anon page, so you corrected it at that time.

Thank you always~

RESPONSE (/RMAP-1/?REPLYTOCOM=305623#RESPOND)

MOON YOUNG-IL (HTTP://JAKE.DOTHOME.CO.KR)
2021-06-21 13:56 (http://jake.dothome.co.kr/rmap-1/#comment-305638)

Yes. ^^;
Removed the word anon.
We've also made some changes to the description, and we've added 3 ways > 4 ways. (Pictures later)
I appreciate it.

RESPONSE (/RMAP-1/?REPLYTOCOM=305638#RESPOND)

LEAVE A COMMENT

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

Comments

이름 *

이메일 *

웹사이트

댓글 작성

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

VMPressure > (http://jake.dothome.co.kr/vmpressure/)