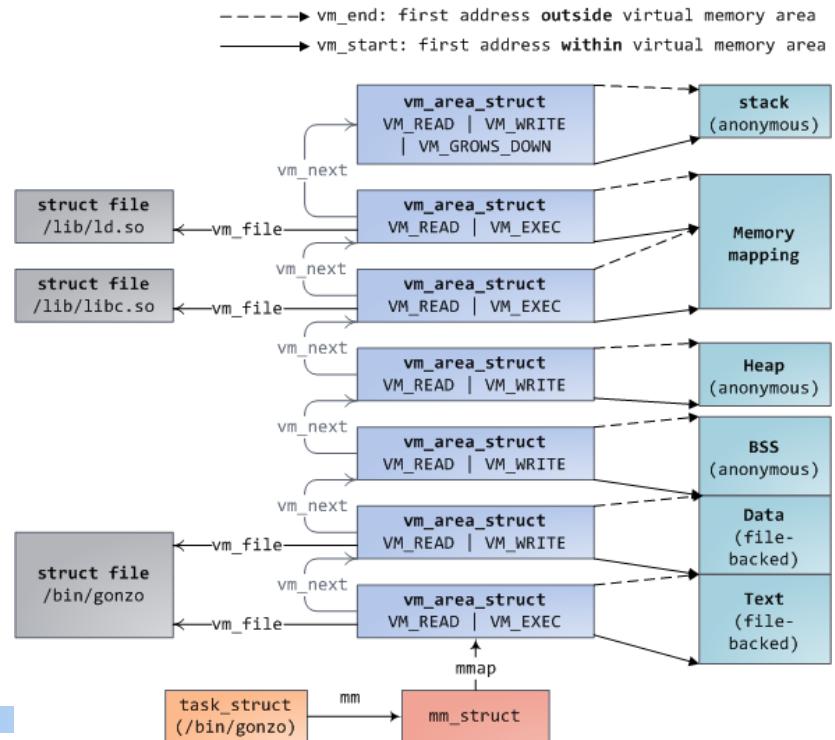
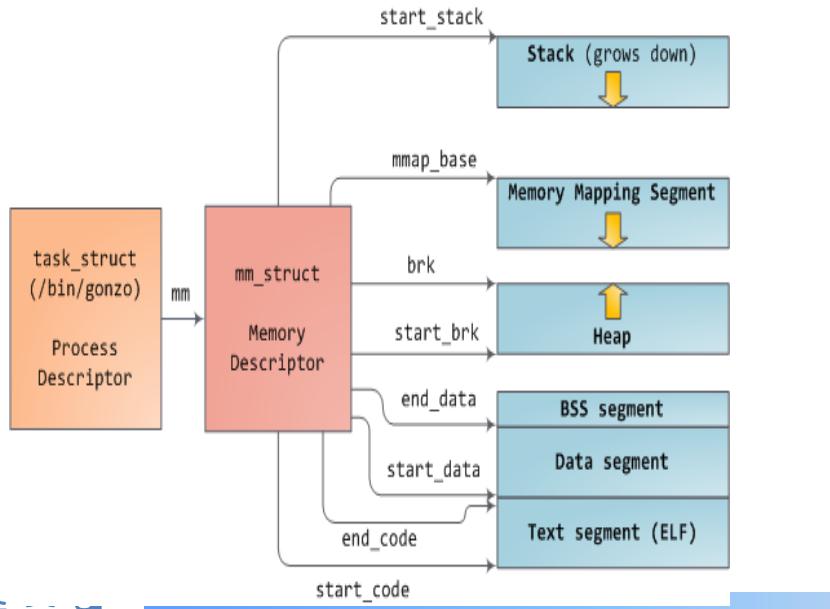


Virtual Process Memory

Son Ju Hyung
tooson9010@gmail.com

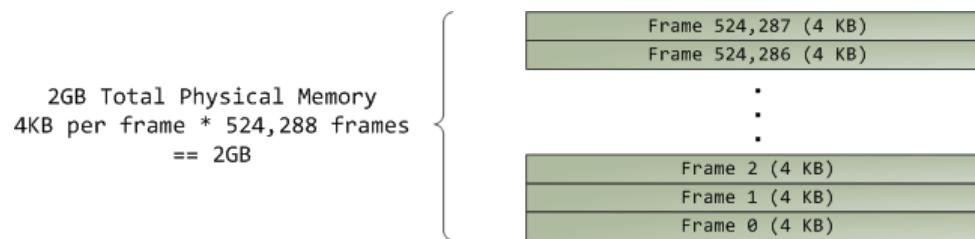
Virtual Process Memory Overview

- linux 는 task_struct 를 통해 process 를 관리하며 task_struct->mm 을 통해 process 가 사용하는 memory 를 관리
 - mm 내부에 각 segment 들의 시작 끝 주소 관리
- process 가 접근하는 virtual memory area 는 vm_area_struct 를 통해 관리
 - 각 vma 는 연속적 공간이며 mmap 을 제외하고 mmap 영역을 제외하고 하나씩 존재(x86) 하며 virtual area 의 시작, 끝을 나타냄
 - starting virtual address 부터 single linked list 로 연결된 구조로 이루어져 있으며 빠른 검색을 위하여 red-black tree 사용(page 별 pte tracking 을 위한 red-black tree 도 존재)
 - vma 를 통해 나타낼 수 있는 address range 는 4K,2M,4M 단위의 page 로 나누어 지며 (x86 기준) 크기는 page 의 배수



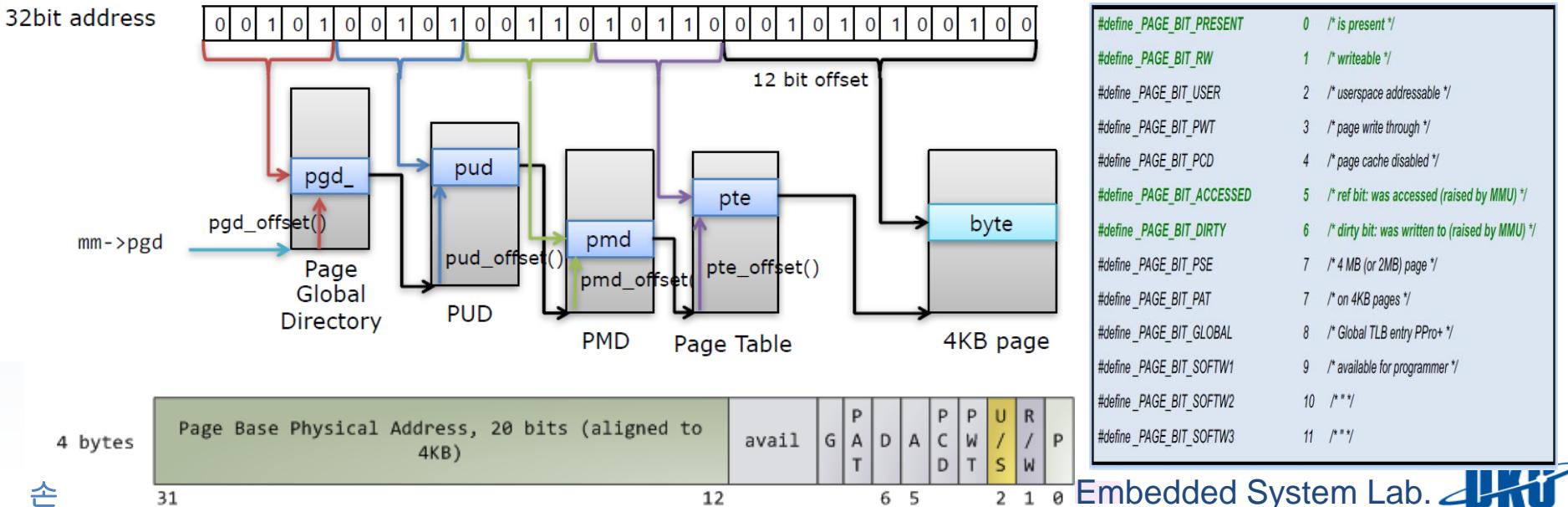
Virtual Process Memory Overview

- linux 는 task_struct 를 통해 process 를 관리하며 task_struct->mm 을 통해 process 가 사용하는 memory 를 관리
 - mm 내부에 각 segment 들의 시작 끝 주소 관리
- process 가 접근하는 virtual memory area 는 vm_area_struct 를 통해 관리
 - 각 vma 는 연속적 공간이며 mmap 을 제외하고 mmap 영역을 제외하고 하나씩 존재(x86) 하며 virtual area 의 시작, 끝을 나타냄
 - starting virtual address 부터 single linked list 로 연결된 구조로 이루어져 있으며 빠른 검색을 위하여 red-black tree 사용(page 별 pte tracking 을 위한 red-black tree 도 존재)
 - vma 를 통해 나타낼 수 있는 address range 는 4K,2M,4M 단위의 page 로 나누어 지며 (x86 기준) 크기는 page 의 배수
 - physical memory 즉 page table 에 지정되는 주소는 4KB 단위로 관리



Virtual Process Memory Overview

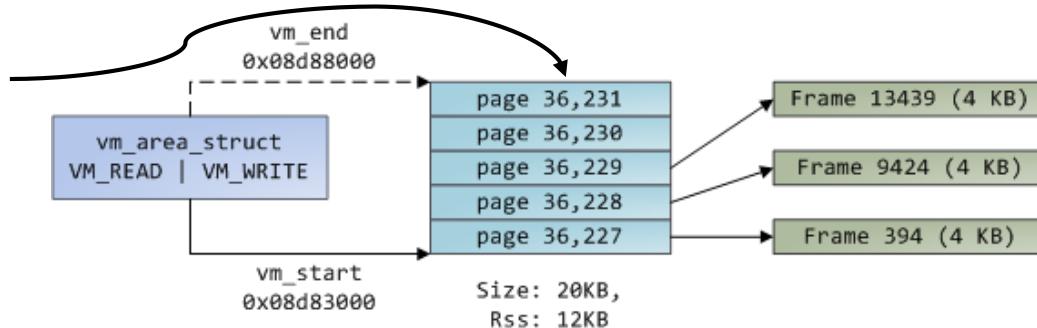
- vma 를 통해 나타내어지는 page 의 실제 할당된 위치는 page table 을 통해 virtual address 에 대한 physical memory address(4KB 단위 align) 를 관리, process 마다 각자의 page table 을 가짐
- page talbe 의 각 entry (PTE) 에는 physical address 이외에도 해당 물리 page frame 의 여러가지 상태를 기술하는 attribute 가 있음
 - P(PAGE_PRESENT) : virtual page에 대한 physical page frame이 memory에 올라와 있는지 검사 0 이면 page fault
 - R/W(PAGE_RW) : physical page frame 에 대한 read/write mode flag. 0 이면 read-only
 - U/S(PAGE_USER) : user/super user mode 에 대한 flag 0이면 kernel 에 의해서만 접근이 가능한 page
 - D(PAGE_DIRTY) : page 가 수정되었으며 backing store 와 다툼을 의미하는 flag, write-back 에 사용
 - A(PAGE_ACCESED) : page reclaim 시 reverse mapping 에 사용



Virtual Process Memory Overview

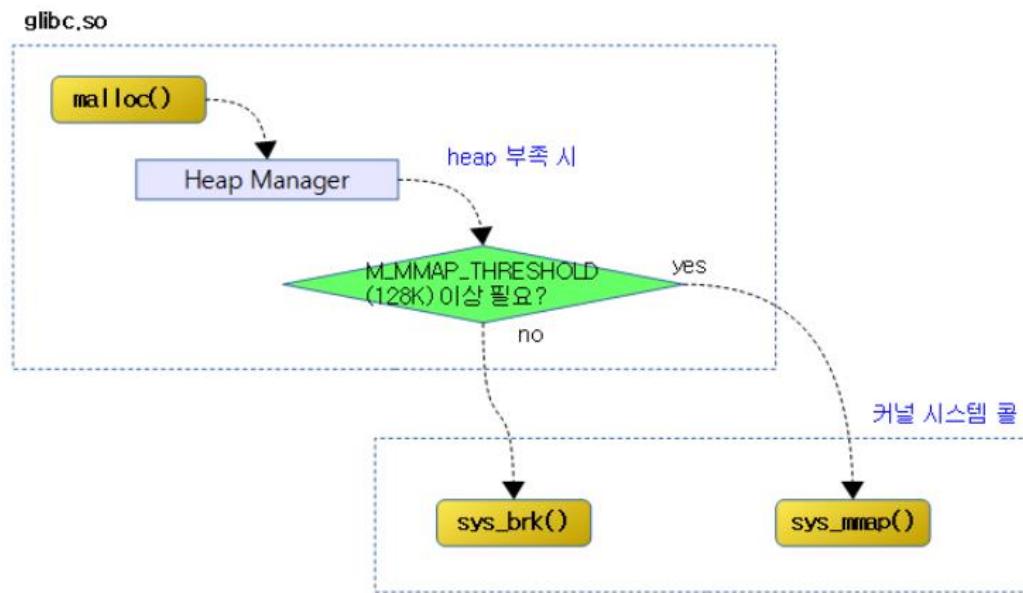
- vma 를 통해 나타내어지는 page 의 실제 할당된 위치는 page table 을 통해 virtual address 에 대한 physical memory address(4KB 단위 align) 를 관리, process 마다 각자의 page table 을 가짐
- page talbe 의 각 entry (PTE) 에는 physical address 이외에도 해당 물리 page frame 의 여러가지 상태를 기술하는 attribute 가 있음
 - P(PAGE_PRESENT) : virtual page에 대한 physical page frame이 memory에 올라와 있는지 검사 0 이면 page fault
 - R/W(PAGE_RW) : physical page frame 에 대한 read/write mode flag. 0 이면 read-only
 - U/S(PAGE_USER) : user/super user mode 에 대한 flag 0이면 kernel 에 의해서만 접근이 가능한 page
 - D(PAGE_DIRTY) : page 가 수정되었으며 backing store 와 달름을 의미하는 flag, write-back 에 사용
 - A(PAGE_ACCESED) : page reclaim 시 reverse mapping 에 사용

- PAGE_PRESENT 가 clear 상태 (swap out / none)
- 접근 시 page fault 발생



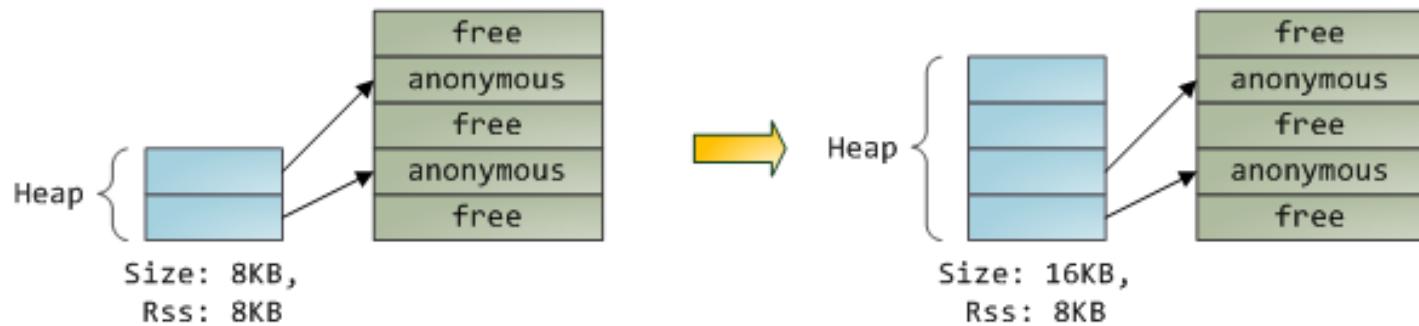
Virtual Process Memory Overview

- kernel 은 vma 와 pte 를 통해 memory 의 page fault, freeing, swaping 등을 관리
- kernel 은 lazy 하기 때문에 vma 는 program 이 시킨 사항들을 의미하며, PTE 는 memory 내에 지금까지 일어난 일들을 의미.
 - malloc() 을 통해 memory 할당
 - Heap Manager 에 의해 현재 할당 요청이 된 memory 가 새로운 vma 를 할당해야 할만큼 큰 할당인지, 아닌지 검사하여 필요하여 128KB 이하일 경우, sys_brk system call 을 통해 heap 에 할당하거나 sys_mmap 을 통해 새로운 vma 생성



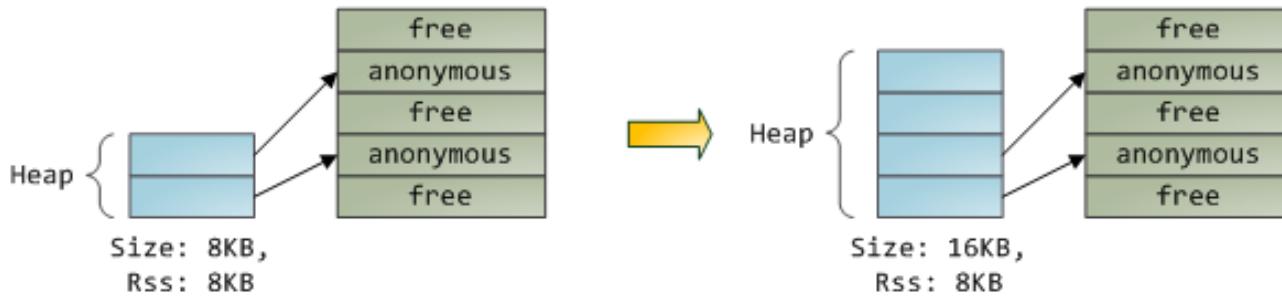
Virtual Process Memory Overview

- kernel 은 vma 와 pte 를 통해 memory 의 page fault, freeing, swaping 등을 관리
- kernel 은 lazy 하기 때문에 vma 는 program 이 시킨 사항들을 의미하며, PTE 는 memory 내에 지금까지 일어난 일들을 의미.
 - malloc() 을 통해 memory 할당 (anonymous page, 8KB 할당 요청 가정)
 - heap 영역에 해당하는 vma 영역의 크기 증가 및 mm update 등 수행
 - vma 를 update 하여 접근 가능한 virtual address 영역 증가
 - 새로 추가된 virtual address 영역에 대해 실제로 physical page frame 은 아직 없음

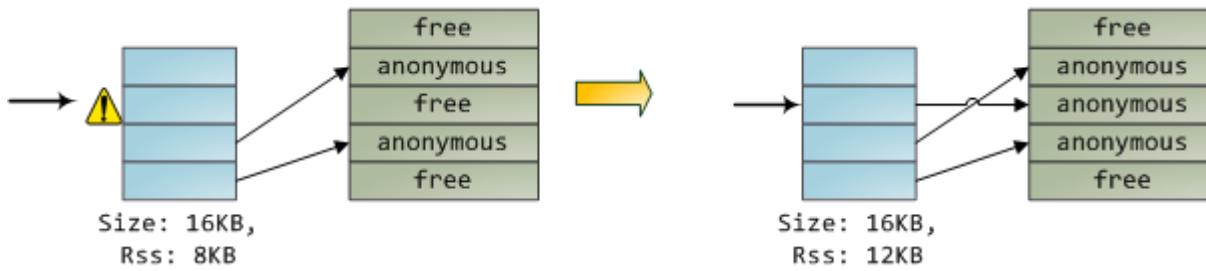


Virtual Process Memory Overview

- kernel 은 vma 와 pte 를 통해 memory 의 page fault, freeing, swaping 등을 관리
- kernel 은 lazy 하기 때문에 vma 는 program 이 시킨 사항들을 의미하며, PTE 는 memory 내에 지금까지 일어난 일들을 의미.
 - malloc() 을 통해 memory 할당 (anonymous page, 8KB 할당 요청 가정)
 - heap 영역에 해당하는 vma 영역의 크기 증가 및 mm update 등 수행

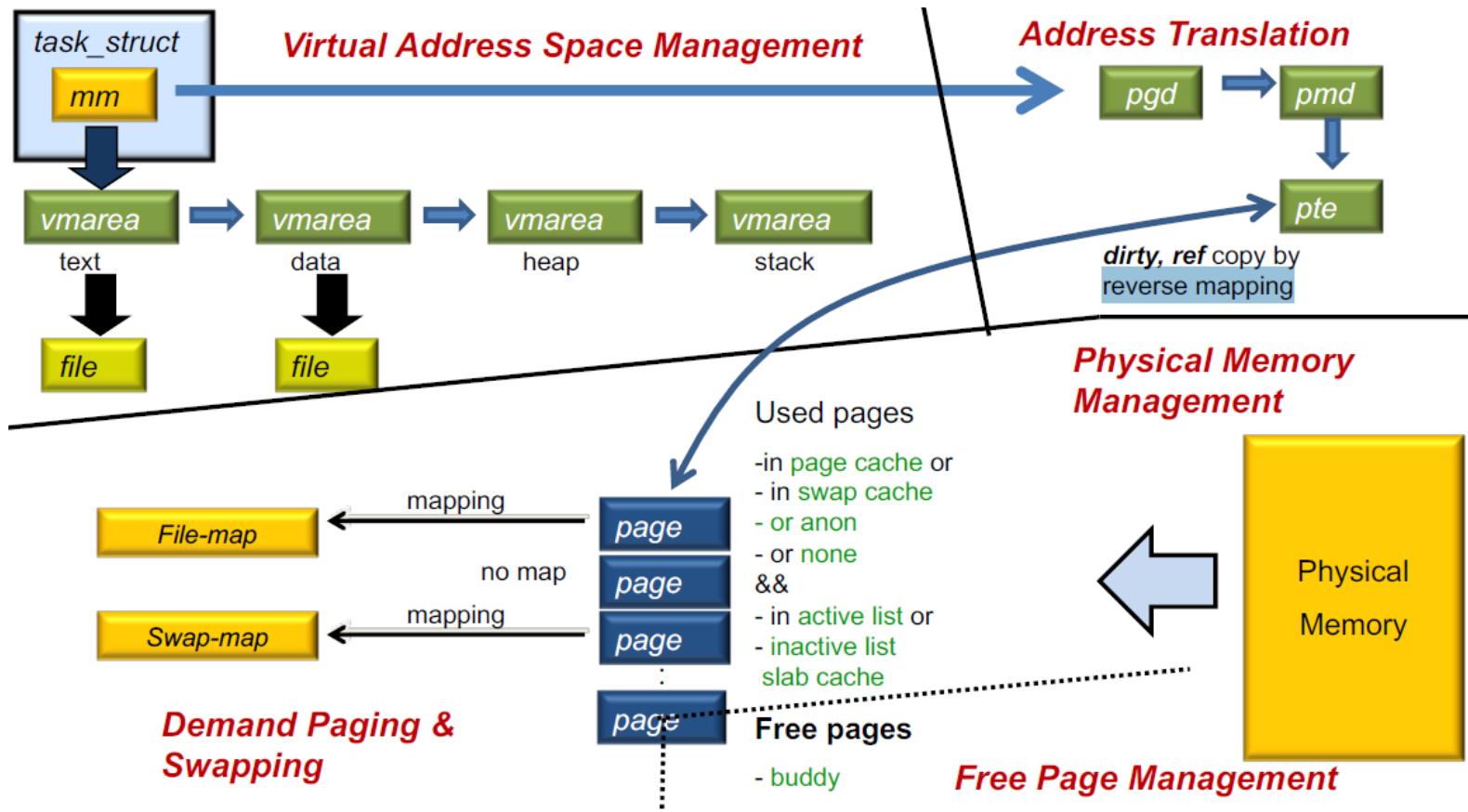


- 추가된 virtual address 영역의 page 에 접근 시, page fault 발생
 - do_page_fault
 - find_vma
 - pte attribute 를 확인하여 PAGE_PRESENT bit clear 확인 및 physical frame 주소 clear 확인
 - do_anonymous_page 를 통해 사용가능 한 physical frame 가져와 pte 및 vma 와 연결
 - PAGE_PRESSET bit 가 clear 되어 있으나, physical frame 주소가 비어있지 않을 경우, do_swap_page 로 수행



Virtual Process Memory

- Virtual Address Space Management



Data Structures

- struct task_struct

```
struct task_struct {  
    volatile long state; /* -1 unrunnable, 0 runnable, >0 stopped */  
    void *stack;  
    ::  
/* scheduler things */  
    ::  
    struct mm_struct *mm, *active_mm;  
    ::  
    struct files_struct *files; /* open file information, not for VM*/  
        /* pointer to struct file * fd_array[NR_OPEN_DEFAULT]; */  
    ::  
/* sig */  
    /* Per-thread vma caching: */  
    struct vmacache vmacache;  
    // 최근에 접근된 vm_area_struct 를 가지고 있는 vmacache  
    // cache size 는 4 개임  
}
```

Data Structures

- struct mm_struct

```
struct mm_struct {
    struct vm_area_struct *mmap;          /* list of VMAs */
    // mm 이 가진 vm area 의 정보를 나타내는 single linked list
    struct rb_root mm_rb;
    // vm_area_struct 와 관련된 rb-tree 를 rb tree 의 root 를 가리킴
    u32 vmacache_seqnum;                /* per-thread vmacache */
    // task_struct 별로 가지고 있는 VMCACHE_SIZE 크기 (vm_area_struct 4개)의 I
    // vmacache 에 해당하는 sequence number
#ifndef CONFIG_MMU
    unsigned long (*get_unmapped_area) (struct file *filp,
                                         unsigned long addr, unsigned long len,
                                         unsigned long pgoff, unsigned long flags);
    // 빈 주소 주간을 찾는 함수
    // (len 에 맞는 target linear free address space 찾음 )
#endif
    unsigned long mmap_base;           /* base of mmap area */
    // memory mapping 영역 start address
    unsigned long mmap_legacy_base;    /* base of mmap area in bottom-up allocations */
    // legacy vm layout 일 경우, mmap 영역의 시작 start address
    unsigned long task_size;          /* size of task vm space */
    // task size 를 의미하며 보통 TASK_SIZE
    //
    // 32bit 의 경우
    //
    // -----
    // | kernel memory (1g) |
    // | ----- |
    // | user memory (3g) | 이 크기가 3g = 0xc0000000 = TASK_SIZE = PAGE_OFFSET
    // -----
    //
    unsigned long highest_vm_end;     /* highest vma end address */
    // vma 중 맨 끝
    pgd_t * pgd;
```

Data Structures

- struct mm_struct

```
atomic_t mm_users;
// mm_struct 를 사용 하는 process 의 수 (e.g. fork 시 증가 )
/** 
 * @mm_count: The number of references to &struct mm_struct
 * (@mm_users count as 1).
 *
 * Use mmgrab()/mmdrop() to modify. When this drops to 0, the
 * &struct mm_struct is freed.
 */
atomic_t mm_count;
// mm_struct 로의 reference 가 있는지에 대한 reference counters
// 즉 mm_users 가 하나 이상이면 mm_count 가 1이다 .
// mm_users 가 0이 되면 mm_count 또한 1 감소 하며 mm_count 가 0 이 되면
// mm_struct 를 free 해 준다 .
atomic_long_t nr_ptes;           /* PTE page table pages */
#if CONFIG_PGTABLE_LEVELS > 2
    atomic_long_t nr_pmds;          /* PMD page table pages */
#endif
int map_count;                  /* number of VMAs */
// mm 에 포함되어 있는 vma 의 개수

spinlock_t page_table_lock;     /* Protects page tables and some counters */
struct rw_semaphore mmap_sem;

struct list_head mmlist;        /* List of maybe swapped mm's. These are globally strung
                                * together off init_mm.mmlist, and are protected
                                * by mmlist_lock
                                */

unsigned long hiwater_rss;    /* High-watermark of RSS usage */
unsigned long hiwater_vm;     /* High-water virtual memory usage */
```

Data Structures

- struct vm_area_struct

```
struct vm_area_struct {
    /* The first cache line has the info for VMA tree walking. */

    unsigned long vm_start;      /* Our start address within vm_mm. */
    // virtual address 시작 주소
    unsigned long vm_end;        /* The first byte after our end address within vm_mm. */
    // virtual address 끝 주소

    /* linked list of VM areas per task, sorted by address */
    struct vm_area_struct *vm_next, *vm_prev;
    // vm_area_struct 의 linear linking pointer

    struct rb_node vm_rb;
    // red black tree 에서의 현재 vm_area_struct 가 속한 node 를
    // rb tree 에서 찾아서 container of 를 vm_area_struct 찾음

    /*
     * Largest free memory gap in bytes to the left of this VMA.
     * Either between this VMA and vma->vm_prev, or between one of the
     * VMAs below us in the VMA rbtree and its ->vm_prev. This helps
     * get_unmapped_area find a free area of the right size.
     */
    unsigned long rb_subtree_gap;
    // 지금 vma 와 바로 전 vma 사이의 gap 또는 그 전 vma 들 사이에서의 gap 를
    // 중 가장 큰 gap 으로 get_unmapped_area 에서 현재 vma 들 사이에 꽂 수 있길
    // 한지 확인 할 때 사용

    /* Second cache line starts here. */

    struct mm_struct *vm_mm;      /* The address space we belong to. */
    // vm_area_struct 가 속한 mm 를 가르키기 위한 back-pointer
    pgprot_t vm_page_prot;       /* Access permissions of this VMA. */
    // 해당 address 에 대한 접근 권한
    unsigned long vm_flags;       /* Flags, see mm.h. */
    // vm region 에 대한 properties
    // e.g. VM_READ, VM_WRITE, VM_EXEC, VM_SHARED : page 내용에 대한 read, write, exec, shared 가능 여부
    // VM_MAYREAD, VM_MAYWRITE, VM_MAYEXEC, VM_MAYSHARE : VM_ 위 플래그들이 설정될 수 있다 ?(mprotect)
    // VM_GROWSDOWN, VM_GROWSUP : stack 은 VM_GROWSDOWN, heapd 은 VM_GROWSUP
    // VM_DONTCOPY : fork 시 해당 vm 영역을 copy 하지 말 것
    // VM_DONTEXPAND : vm 영역 rmremap 등으로 확장 불가능
```

Data Structures

- struct vm_area_struct

```
struct {
    struct rb_node rb;
    // left subtree 를 중에서의 vm_end max 값을 가지는 node
    unsigned long rb_subtree_last;
    // left subtree 를 중에서의 vm_end max 값
} shared;
// 옛날엔 prio_tree 로 관리 되었지만, rb_tree 로 관리되도록 patch 됨
/*
 * A file's MAP_PRIVATE vma can be in both i_mmap tree and anon_vma
 * list, after a COW of one of the file pages. A MAP_SHARED vma
 * can only be in the i_mmap tree. An anonymous MAP_PRIVATE, stack
 * or brk vma (with NULL file) can only be in an anon_vma list.
 */
struct list_head anon_vma_chain; /* Serialized by mmap_sem & * page_table_lock */
// heap, stack, vma chain
// swap 되기 전의 dirty anon, dirty data 관리

struct anon_vma *anon_vma; /* Serialized by page_table_lock */
// anonymous page COW handling, shared page 관리
// reverse mapping 관련

/* Function pointers to deal with this struct. */
const struct vm_operations_struct *vm_ops;
// demand paging을 위한 open, close, mmap 함수들

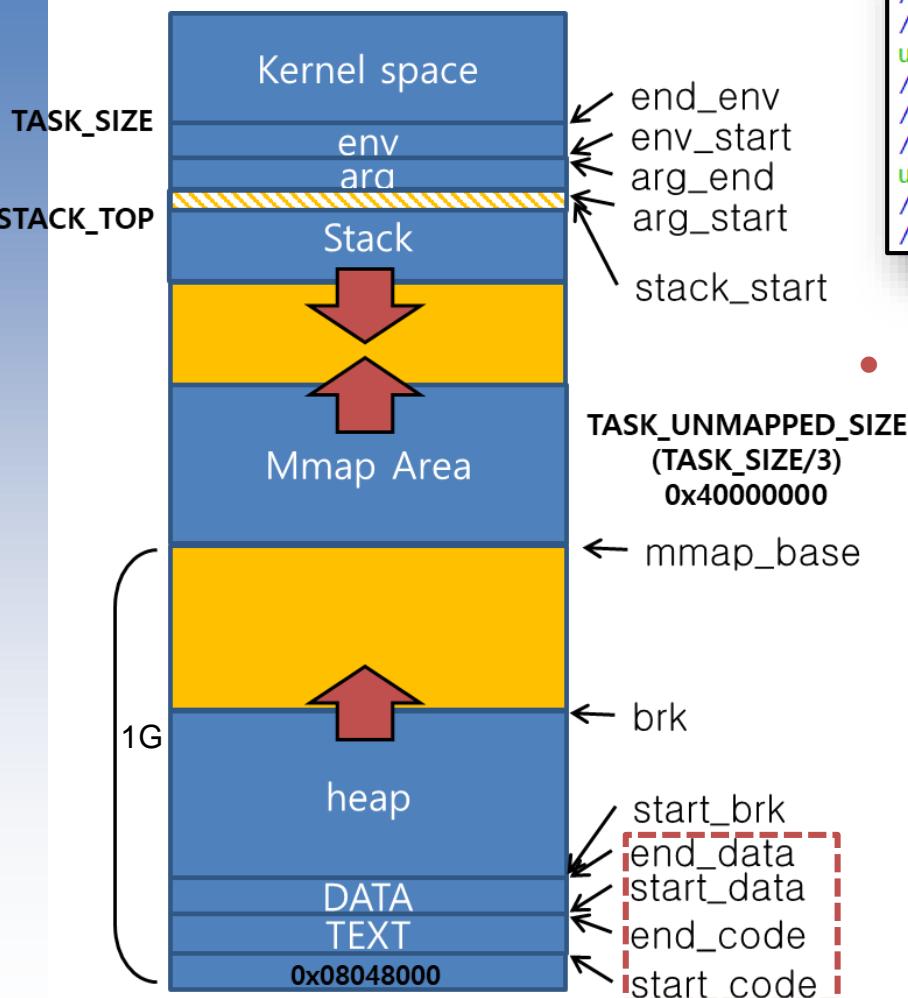
/* Information about our backing store: */
unsigned long vm_pgoff; /* Offset (within vm_file) in PAGE_SIZE units */
// file mapping 시에 전체 file 이 mapping 된 것이라면 0 으로, 부분만 mapping
// 된 것이라면 그 offset 을 의미 (page 수 기준)
struct file * vm_file; /* File we map to (can be NULL). */
// virtual address space 에 mapping 된 struct file
void * vm_private_data; /* was vm_pte (shared mem) */

#ifndef CONFIG_MMU
    struct vm_region *vm_region; /* NOMMU mapping region */
#endif
#ifndef CONFIG_NUMA
    struct mempolicy *vm_policy; /* NUMA policy for the VMA */
#endif
    struct vm_userfaultfd_ctx vm_userfaultfd_ctx;
};
```

Data Structures

- struct anon_vma, struct anon_vma_chain
 - struct vm_operations_struct
 - struct address_space_operations
 - struct rb_node, struct rb_root
 - struct vmcache
- ...
- will visit again

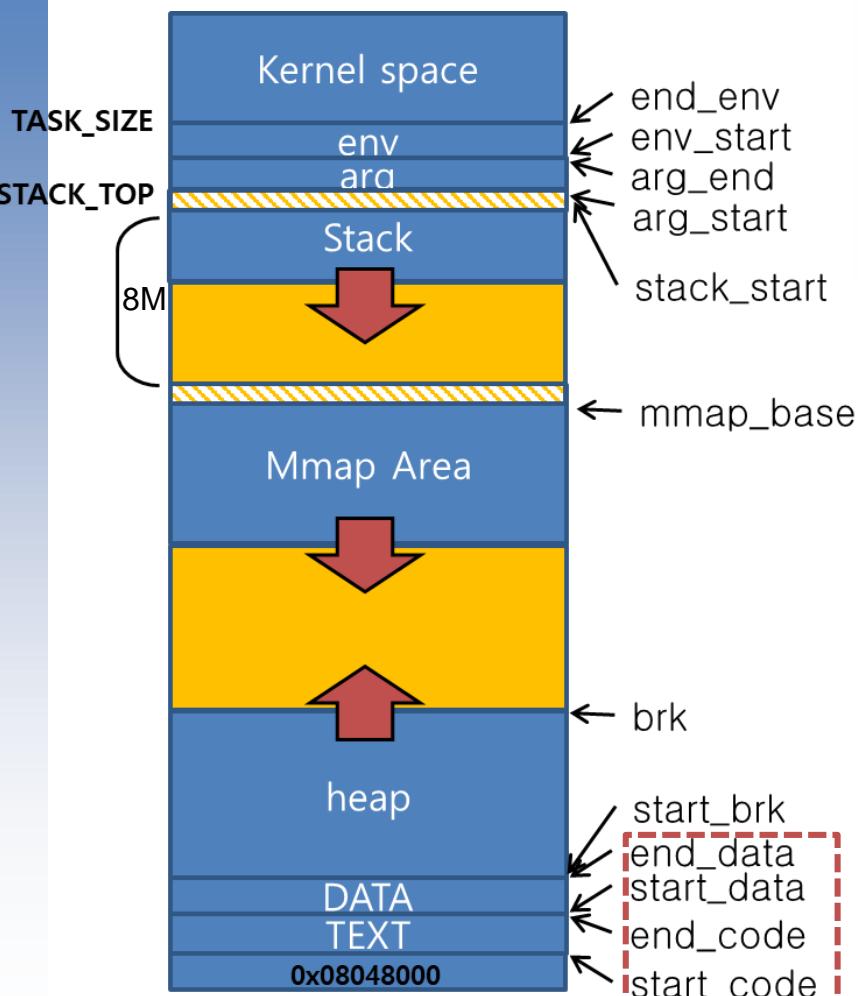
Virtual Process Address Space Layout



```
unsigned long start_code, end_code, start_data, end_data;  
// start_code, end_code : code 영역 start ~ end address  
// start_data, end_data : data 영역 start ~ end address  
unsigned long start_brk, brk, start_stack;  
// start_brk : heap start address  
// brk : heap data의 현재 end address  
// start_stack : stack start address  
unsigned long arg_start, arg_end, env_start, env_end;  
// arg_start, arg_end : argument list 영역 start ~ end address  
// env_start, env_end : environment 영역 start ~ end address
```

- Classical virtual memory layout
 - mm_struct 를 통해 virtual address space 의 각 segment range 나타냄.
 - data, code range 는 ELF binary 가 map 된 후 수정 안됨
 - architecture 별로 자신만의 layout 관련 함수 설정 가능
 - layout 직접 설정(HAVE_ARCH_PICK_MMAP_LAYOUT)
 - arch_pick_mmap_layout
 - mmap 시 할당 위치 설정(HAVE_ARCH_UNMAPPED_AREA)
 - arch_get_unmapped_area
 - text 영역 시작 전 영역 일부 ERR code reference 로 사용
 - mmap 시작 주소는 TASK_SIZE/3 으로 고정
 - address space randomization(PF_RANDOM)

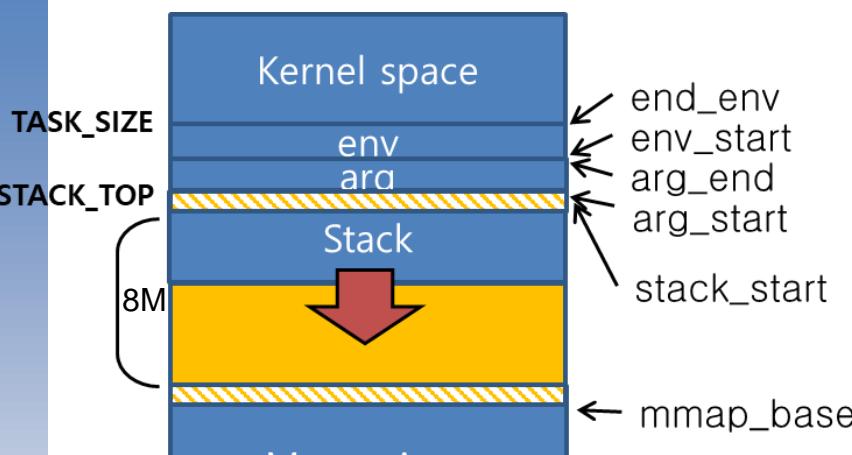
Virtual Process Address Space Layout



```
unsigned long start_code, end_code, start_data, end_data;
// start_code, end_code : code 영역 start ~ end address
// start_data, end_data : data 영역 start ~ end address
unsigned long start_brk, brk, start_stack;
// start_brk : heap start address
// brk : heap data의 현재 end address
// start_stack : stack start address
unsigned long arg_start, arg_end, env_start, env_end;
// arg_start, arg_end : argument list 영역 start ~ end address
// env_start, env_end : environment 영역 start ~ end address
```

- Classical Virtual memory layout
 - mm_struct 를 통해 virtual address space 의 각 segment range 나타냄.
 - data, code range 는 ELF binary 가 map 된 후 수정 안됨
 - architecture 별로 자신만의 layout 관련 함수 설정 가능
 - layout 직접 설정(HAVE_ARCH_PICK_MMAP_LAYOUT)
 - arch_pick_mmap_layout
 - mmap 시 할당 위치 설정(HAVE_ARCH_UNMAPPED_AREA)
 - arch_get_unmapped_area
 - text 영역 시작 전 영역 일부 ERR code reference 로 사용
 - mmap 시작 주소는 TASK_SIZE/3 으로 고정
 - address space randomization(PF_RANDOM)
- Virtual memory layout
 - stack size 고정 및 stack 과 mmap 사이 safety gap

Virtual Process Address Space Layout



```
unsigned long start_code, end_code, start_data, end_data;  
// start_code, end_code : code 영역 start ~ end address  
// start_data, end_data : data 영역 start ~ end address  
unsigned long start_brk, brk, start_stack;  
// start_brk : heap start address  
// brk : heap data 의 현재 end address  
// start_stack : stack start address  
unsigned long arg_start, arg_end, env_start, env_end;  
// arg_start, arg_end : argument list 영역 start ~ end address  
// env_start, env_end : environment 영역 start ~ end address
```

- Classical Virtual memory layout
 - mm_struct 를 통해 virtual address space 의 각 segment range 나타냄

root@son:/home/son/workspace/git/embedded/Professional-Linux-Kernel-Architecture/linux-4.11# cat /proc/2725/limits	Limit	Soft Limit	Hard Limit	Units
	Max cpu time	unlimited	unlimited	seconds
	Max file size	unlimited	unlimited	bytes
	Max data size	unlimited	unlimited	bytes
	Max stack size	8388608	8388608	8MB
	Max core file size	0	unlimited	bytes
	Max resident set	unlimited	unlimited	bytes
	Max processes	126515	126515	processes
	Max open files	10032	10032	files
	Max locked memory	65536	65536	bytes
	Max address space	unlimited	unlimited	bytes
	Max file locks	unlimited	unlimited	locks
	Max pending signals	126515	126515	signals
	Max msgqueue size	819200	819200	bytes
	Max nice priority	0	0	
	Max realtime priority	0	0	
	Max realtime timeout	unlimited	unlimited	us

Virtual Process Address Space Layout

- Creating the Layout
 - load_elf_binary 함수를 통해 ELF binary 가 load 될 때, task layout 결정(exec 함수에서 사용)
 - load_elf_binar
 - /proc/sys/kernel/randomize_va_space 설정 검사 및 personality system call 로 인한 process->private 조건 설정 여부 검사
 - /proc/sys/kernel/legacy_va_layout (책) -> /proc/sys/vm/legacy_va_layout
 - setup_new_exec
 - arch_pick_mmap_layout 함수가 arch 별로 지정 여부에 따라 classic layout, modified layout 결정
 - setup_arg_pages
 - stack 의 vm_area_struct 할당 및 설정

```
static int load_elf_binary(struct linux_binprm *bprm)
{
...
if (!(current->personality & ADDR_NO_RANDOMIZE) && randomize_va_space)
    current->flags |= PF_RANDOMIZE;
// personality system call 를 통해 randomize 하지 말도록 설정한 상태가
// 아니고, randomize_va_space 가 sysfs 를 통해 설정되어 있다면 VA layout
// 생성 시 randomize 하도록 task_struct 에 flag 설정(PF_RANDOMIZE)

setup_new_exec(bprm);
install_exec_creds(bprm);

/* Do this so that we can load the interpreter, if need be. We will
   change some of these later */
retval = setup_arg_pages(bprm, randomize_stack_top(STACK_TOP),
                        executable_stack);
// 지정된 위치에 stack 을 생성 하며 arch specific 한 STACK_TOP 을 넘겨 주며,
// PF RANDOMIZE 설정 시, top 위치에 대해 randomize 수행

...
}
```

```
void setup_new_exec(struct linux_binprm *bprm)
{
    arch_pick_mmap_layout(current->mm);
    ...
}
```

```
#if defined(CONFIG_MMU) && !defined(HAVE_ARCH_PICK_MMAP_LAYOUT)
void arch_pick_mmap_layout(struct mm_struct *mm)
{
    mm->mmap_base = TASK_UNMAPPED_BASE;
    mm->get_unmapped_area = arch_get_unmapped_area;
}
#endif
```

Virtual Process Address Space Layout

- Creating the Layout
 - load_elf_binary 함수를 통해 ELF binary 가 load 될 때, task layout 결정(exec 함수에서 사용)
 - load_elf_binar
 - /proc/sys/kernel/randomize_va_space 설정 검사 및 personality system call 로 인한 process->private 조건 설정 여부 검사
 - /proc/sys/kernel/legacy_va_layout (책) -> /proc/sys/vm/legacy_va_layout
 - setup_new_exec
 - arch_pick_mmap_layout 함수가 arch 별로 지정 여부에 따라 classic layout, modified layout 결정
 - setup_arg_pages
 - stack 의 vm_area_struct 할당 및 설정

```
static int load_elf_binary(struct linux_binprm *bprm)
{
    ...
    if (!(current->personality & ADDR_NO_RANDOMIZE) && randomize_va_space)
        current->flags |= PF_RANDOMIZE;
    // personality system call 를 통해 randomize 하지 말도록 설정한 상태가
    // 아니고 ,randomize_va_space 가 sysfs 를 통해 설정되어 있다면 VA layout
    // 생성 시 randomize 하도록 task_struct 에 flag 설정(PF_RANDOMIZE)
    setup_new_exec(bprm);
    install_exec_creds(bprm);

    /* Do this so that we can load the interpreter, if need be. We will
     * change some of these later */
    retval = setup_arg_pages(bprm, randomize_stack_top(STACK_TOP),
                           executable_stack);
    // 지정된 위치에 stack 을 생성 하며 arch specific 한 STACK_TOP 을 넘겨 주며 ,
    // PF RANDOMIZE 설정 시 , top 위치에 대해 randomize 수행
    ...
}
```

```
void setup_new_exec(struct linux_binprm *bprm)
{
    arch_pick_mmap_layout(current->mm);
    ...
}
```

```
// new process 의 vma layout 생성 시마다 호출되어 layout type 결정
void arch_pick_mmap_layout(struct mm_struct *mm)
{
    unsigned long random_factor = 0UL;
    // PF_RANDOMIZE 설정 되어 있다면 즉 vma 에 randomize 적용할꺼면
    // 각 architecture 마다 제공하는 random generator 함수 사용하여
    // random long sized number 생성
    if (current->flags & PF_RANDOMIZE)
        random_factor = arch_mmap_rnd();
    // 상위 주소로 자라는 mmap layout 의 시작 주소에 random 값 추가하여
    // legacy mmap start area 초기화
    mm->mmap_legacy_base = TASK_UNMAPPED_BASE + random_factor;
    // 이제 어떤 vm layout 을 할지 mmap_is_legacy 함수를 통해 결정
    if (mmap_is_legacy()) {
        mm->mmap_base = mm->mmap_legacy_base;
        // legacy 일 경우 , mmap 시작 주소를 TASK_UNMAPPED_BASE 로 고정 위치
        mm->get_unmapped_area = arch_get_unmapped_area;
        // free area 찾는 함수 설정
    } else {
        mm->mmap_base = mmap_base(random_factor);
        // mmap_base 함수를 통해 고정된 TASK_SIZE - stack 크기 - random 값으로
        // mmap 주소 설정
        mm->get_unmapped_area = arch_get_unmapped_area_topdown;
    }
}
```

Virtual Process Address Space Layout

Creating the Layout

- load_elf_binary 함수를 통해 ELF binary 가 load 될 때, task layout 결정(exec 함수에서 사용)
- arch_pick_mmap_layout
 - 어떤 layout 을 선택할 지는 mmap_is_legacy 함수를 통해 결정 됨.
 - personality system call 을 통해 legacy 로 설정하도록 한적이 있거나, stack size 가 고정인지, /proc 을 통해 따로 설정 된 값이 있는지 가져옴
 - 결정된 layout 에 따라 mmap 시작 위치 설정 및 빈 주소 구간을 찾는 함수를 다르게 설정
 - legacy 로 하도록 설정 시 mmap 시작 주소 위치 고정
 - modified layout 은... mmap 시작 주소 위치가 mmap_base 함수를 통해 설정
 - TASK_SIZE – 스택 크기 – random 값

```
unsigned int personality;
// linux 에서는 process 마다 vm layout, vaddr limit address, 등 을 다르게
// 설정 할 수 있음. 이 값은 personality system call 을 통해 설정 가능
// e.g. ADDR_COMPAT_LAYOUT : legacy virtual address 로 되도록 설정 (mmap 위로 )
// ADDR_NO_RANDOMIZE : address-space-layout Randomize 하지 않음
```

```
static int load_elf_binary(struct linux_binprm *bprm)
{
    ...
    if (!(current->personality & ADDR_NO_RANDOMIZE) && randomize_va_space)
        current->flags |= PF_RANDOMIZE;
    // personality system call 을 통해 randomize 하지 말도록 설정한 상태가
    // 아니고 ,randomize_va_space 가 sysfs 를 통해 설정되어 있다면 VA layout
    // 생성 시 randomize 하도록 task_struct 에 flag 설정 (PF_RANDOMIZE)

    setup_new_exec(bprm);
    install_exec_creds(bprm);

    /* Do this so that we can load the interpreter, if need be. We will
     * change some of these later */
    retval = setup_arg_pages(bprm, randomize_stack_top(STACK_TOP),
                           executable_stack);
    // 지정된 위치에 stack 을 생성 하며 arch specific 한 STACK_TOP 을 넘겨 주며 ,
    // PF RANDOMIZE 설정 시 , top 위치에 대해 randomize 수행

    ...
}
```

```
void setup_new_exec(struct linux_binprm *bprm)
{
    arch_pick_mmap_layout(current->mm);
    ...
}
```

```
// new process 의 vma layout 생성 시마다 호출되어 layout type 결정
void arch_pick_mmap_layout(struct mm_struct *mm)
{
    static int mmap_is_legacy(void)
    {
        if (current->personality & ADDR_COMPAT_LAYOUT)
            return 1;
        // personality system call 을 통해 legacy layout
        // 으로 하라고 설정 되어 있는지 검사
        if (rlimit(RLIMIT_STACK) == RLIM_INFINITY)
            return 1;
        // tsk->signal->rlim[limit].rlim_cur 에 설정된 rlimit
        // 배열 요소 중 3 번째 entry 인 stack size 관련 soft limit
        // 값이 고정되어 있는지 검사 .
        // legacy layout 은 stack size 가 지정되어 있지 않고
        // default layout 은 아래로 자라는 mmap 영역을 짐벌하지 않기 위해
        // stack size 가 고정이기 때문에 RLIM_INFINITY 로 설정되어 있다면
        // legacy layout 입
        return sysctl_legacy_va_layout;
        // /proc/sys/vm/legacy_va_layout 에 설정된 값을 반환
    }
    mm->get_unmapped_area = arch_get_unmapped_area_topdown;
}
```

Virtual Process Address Space Layout

- Creating the Layout
 - load_elf_binary 함수를 통해 ELF binary 가 load 될 때, task layout 결정(exec 함수에서 사용)
 - arch_pick_mmap_layout
 - 어떤 layout 을 선택할 지는 mmap_is_legacy 함수를 통해 결정 됨.
 - personality system call 을 통해 legacy 로 설정하도록 한적이 있거나, stack size 가 고정인지, /proc 을 통해 따로 설정 된 값이 있는지 가져옴
 - 결정된 layout 에 따라 mmap 시작 위치 설정 및 빈 주소 구간을 찾는 함수를 다르게 설정
 - legacy 로 하도록 설정 시 mmap 시작 주소 위치 고정
 - modified layout 은... mmap 시작 주소 위치가 mmap_base 함수를 통해 설정
 - TASK_SIZE – 스택 크기 – random 값

```
static int load_elf_binary(struct linux_binprm *bprm)
{
    ...
    if (!(current->personality & ADDR_NO_RANDOMIZE) && randomize_va_space)
        current->flags |= PF_RANDOMIZE;
    // personality system call 을 통해 randomize 하지 말도록 설정한 상태가
    // 아니고 ,randomize_va_space 가 sysfs 를 통해 설정되어 있다면 VA layout
    // 생성시 randomize 하도록 task_struct 에 flag 설정(PF_RANDOMIZE)

    setup_new_exec(bprm);
    install_exec_creds(bprm);

    /* Do this so that we can load the interpreter, if need be. We will
     * change some of these later */
    retval = setup_arg_pages(bprm, randomize_stack_top(STACK_TOP),
                           executable_stack);
    // 지정된 위치에 stack 을 생성 하며 arch specific 한 STACK_TOP 을 넘겨 주며 ,
    // PF RANDOMIZE 설정 시 , top 위치에 대해 randomize 수행

    ...
}
```

```
void setup_new_exec(struct linux_binprm *bprm)
{
    arch_pick_mmap_layout(current->mm);
    ...
}
```

```
// new process 의 vma layout 생성 시마다 호출되어 layout type 결정
void arch_pick_mmap_layout(struct mm_struct *mm)
{
    unsigned long random_factor = 0UL;
    // PF_RANDOMIZE 설정 되어 있다면 즉 vma 에 randomize 적용 할꺼면
    // 각 architecture 마다 제공하는 random generator 함수 사용하여
    // random long sized number 생성
    if (current->flags & PF_RANDOMIZE)
        random_factor = arch_mmap_rnd();
    // 상위 주소로 자라는 mmap layout 의 시작 주소에 random 값 추가하여
    // legacy mmap start area 초기화
    mm->mmap_legacy_base = TASK_UNMAPPED_BASE + random_factor;
    // 이제 어떤 vm layout 을 할지 mmap_is_legacy 함수를 통해 결정
    if (mmap_is_legacy()) {
        mm->mmap_base = mm->mmap_legacy_base;
        // legacy 일 경우 , mmap 시작 주소를 TASK_UNMAPPED_BASE 로 고정 위치
        mm->get_unmapped_area = arch_get_unmapped_area;
        // free area 찾는 함수 설정
    } else {
        mm->mmap_base = mmap_base(random_factor);
        // mmap_base 함수를 통해 고정된 TASK_SIZE - stack 크기 - random 값으로
        // mmap 주소 설정
        mm->get_unmapped_area = arch_get_unmapped_area_topdown;
    }
}
```

Virtual Process Address Space Layout

- Creating the Layout
 - load_elf_binary 함수를 통해 ELF binary 가 load 될 때, task layout 결정(exec 함수에서 사용)
 - arch_pick_mmap_layout
 - 어떤 layout 을 선택할 지는 mmap_is_legacy 함수를 통해 결정 됨.
 - personality system call 을 통해 legacy 로 설정하도록 한적이 있거나, stack size 가 고정인지, /proc 을 통해 따로 설정 된 값이 있는지 가져옴
 - 결정된 layout 에 따라 mmap 시작 위치 설정 및 빈 주소 구간을 찾는 함수를 다르게 설정
 - legacy 로 하도록 설정 시 mmap 시작 주소 위치 고정
 - modified layout 은... mmap 시작 주소 위치가 mmap_base 함수를 통해 설정
 - TASK_SIZE – 스택 크기 – random 값

```
static int load_elf_binary(struct linux_binprm *bprm)
{
    ...
    if (!(current->personality & ADDR_NO_RANDOMIZE) && randomize_va_space)
        current->flags |= PF_RANDOMIZE;
    // personality system call を 통해 randomize 하지 말도록 설정한 상태가
    // 아니고, randomize_va_space 가 sysfs 를 통해 설정되어 있다면 VA layout
    // 생성 시 randomize 하도록 task_struct 에 flag 설정 (PF_RANDOMIZE)
    setup_new_exec(bprm);
    install_exec_creds(bprm);

    /* Do this so that we can load the interpreter, if need be. We will
     * change some of these later */
    retval = setup_arg_pages(bprm, randomize_stack_top(STACK_TOP),
                           executable_stack);
    // 지정된 위치에 stack 을 생성하며 arch specific 한 STACK_TOP 을 넘겨 주며 ,
    // PF RANDOMIZE 설정 시 , top 위치에 대해 randomize 수행
    ...
}
```

```
void setup_new_exec(struct linux_binprm * bprm)
{
    arch_pick_mmap_layout(current->mm);
    ...
}
```

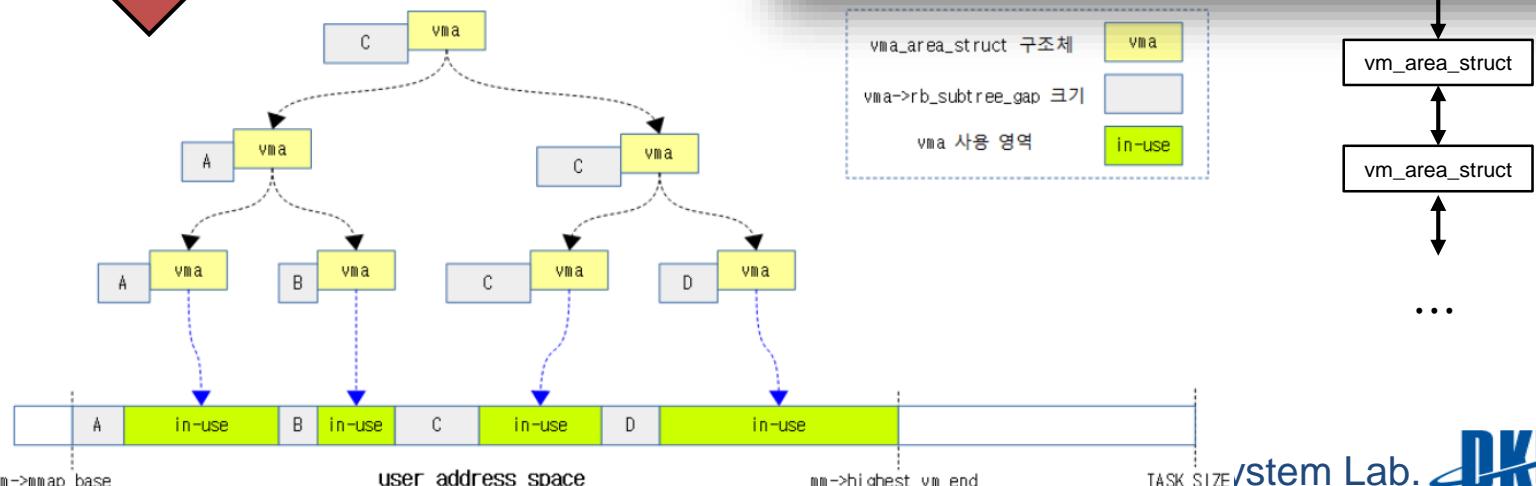
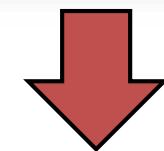
```
// new process 정
void arch_pick_mmap_base(unsigned long rnd)
{
    unsigned long gap = rlimit(RLIMIT_STACK);
    // default layout 일 경우 , stack size 가 고정임
    if (gap < MIN_GAP)
        gap = MIN_GAP;
    // MIN_GAP : 128*1024*1024UL + stack_maxrandom_size()
    // stack 크기 가 최소 128 MB 는 되도록 설정
    else if (gap > MAX_GAP)
        gap = MAX_GAP;
    // MAX_GAP : (TASK_SIZE/6*5)
    // stack 의 최대 크기 제한
    return PAGE_ALIGN(TASK_SIZE - gap - rnd);
    // stack 의 크기를 통해 mmap 의 시작 위치 구함
    // | kernel|
    // -----> TASK_SIZE
    mm->mmap_base = rnd + gap;
    // stack | + | stack size | gap
    // | | | + | |
    // |-----| + |-----|
    mm->get_random = rnd;
    // free
    // |-----| + |-----|
    // |-----| + |-----|
    // mma |-----|
    // mma |-----|
    mm->get_random = rnd;
    // ...
}
}
```

Management of Regions

- 각 region 은 vm_area_struct 로 관리 하며 process 별 vm_area_struct 를 mm_struct 에서 관리
 - mmap 을 통해 signle linked list 로 연결 및
 - mm_rb 를 통해 다른 vma 와 red-black tree 로 연결
- red-black tree 를 통해 접근하려는 address 에 대하여 log(N) 시간에 vma 검색 가능.
 - red-black tree 를 확장한 augmented-rbtree(interval tree) 를 통해 mapping 되지 않은 빈 영역을 찾을 때 subtree 들이 관리하는 gap 영역에 대한 추가 정보를 관리하여 최적화

```
struct mm_struct {
    struct vm_area_struct *mmap;           /* list of VMAs */
    // mm 이 가진 vm area 의 정보를 나타내는 single linked list
    struct rb_root mm_rb;
    // vm_area_struct 와 관련된 rb- tree 로 rb tree 의 root 를 가리킴
    u32 vmacache_seqnum;                  /* per-thread vmacache */
    // task_struct 별로 가지고 있는 VMCACHE_SIZE 크기 (vm_area_struct 4개) 외
    // vmacache 에 해당하는 sequence number
}
```

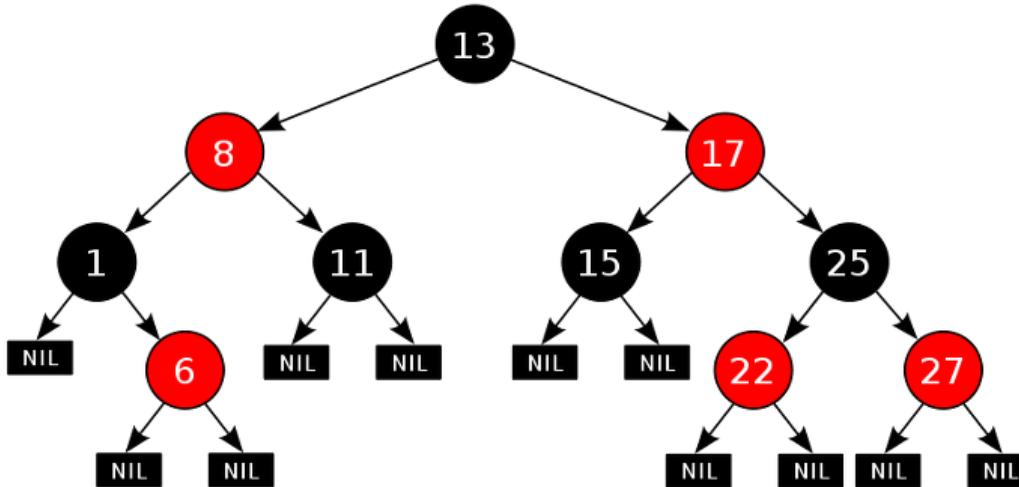
```
struct vm_area_struct {
    unsigned long vm_start;
    // virtual address 시작 주소
    unsigned long vm_end;
    // virtual address 끝 주소
    ...
    unsigned long rb_subtree_gap;
    // 지금 vma 와 바로 전 vma 사이의 gap 또는 그 전 vma 를 사이에서의 gap
    // 중 가장 큰 gap 으로 get_unmapped_area 에서 size 에 맞는 빈 영역을
    // 찾을 때 활용
    // augmented rb tree 에서 관리되는 정보
    struct mm_struct *vm_mm;             /* The address space we belong to. */
    // vm_area_struct 가 속한 mm 를 가르키기 위한 back-pointer
    struct vm_area_struct *vm_next, *vm_prev;
    // vm_area_struct 의 linear linking pointer
}
```



Appendix – Red Black Tree

- classic red-black tree

- BST 기반으로 최대 2개의 child 를 가지며 left child 는 key 값보다 작은 tree, right 는 큰 tree 구조
- self-balancing tree이며 최악의 경우에도 $\log(N)$ 의 시간에 검색/삽입/삭제 수행 anticipatory, deadline, CFQ, I/O scheduler, vma 관리, reverse mapping 등에서 사용
- linux에서 rbtree를 사용하는 곳이 많으며 각각의 필요에 따라 비교하는 알고리즘 또한 다르기 때문에 insert/search 함수 구현 및 locking을 개발자에게 넘김
- Rule
 - 모든 node는 red 또는 black
 - root node는 black, new node는 red
 - red node의 자식이 red가 될 수는 없음. 다른 경우는 다 가능
 - root에서 leaf까지의 black node의 개수는 동일



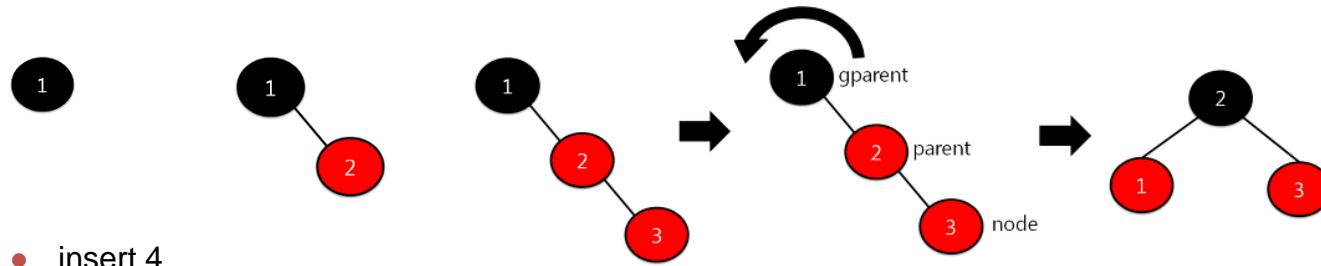
Appendix – Red Black Tree

Algorithm

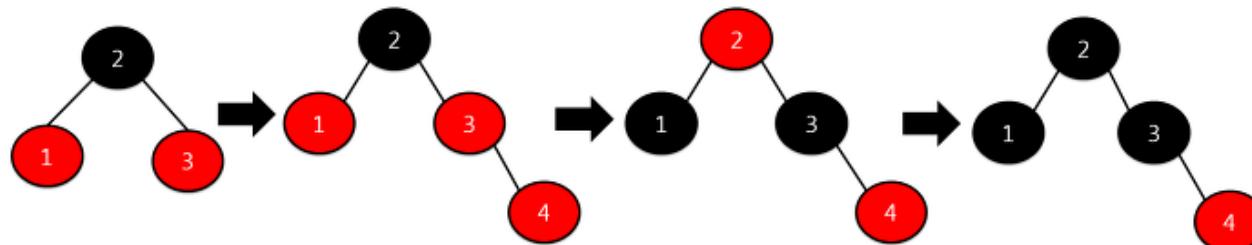
- case by Rule 4. – red node 는 red node 를 자식으로 가질 수 없음
 - 1. uncle 없고, new 가 parent 의 left child
 - 1. 2 와 같으며 방향 반대
 - 2. uncle 없고, new 가 parent 의 right child
 - 1. gparent 위치 기준으로 right rotate
 - 2. parent 위치에서 child 위치로 내려가는 놈은 red 로 설정 & child 위치에서 parent 위치로 올라가는 놈은 black 으로 설정
 - 3. Rule 1 검사
 - 3. uncle 있고, uncle 이 red
 - 1. gparent, uncle, parent 현재 색의 반대로 바꿈
 - 2. Rule 1 검사
 - 4. uncle 있고, uncle 이 black 이며 new 가 parent 의 left child
 - 1. 5와 같으며 방향 반대
 - 5. uncle 있고, uncle 이 black 이며 new 가 parent 의 right child
 - 1. gparent 위치 기준으로 left rotate
 - 2. parent 위치에서 child 위치로 내려가는 놈은 red 로 설정 & child 위치에서 parent 위치로 올라가는 놈은 black 으로 설정
 - 3. parent 위치에 있던 node 가 gparent 위치로 올라가며 left child 를 버리고 버려진 left child 는 uncle 위치로 내려가게 된 gparent 위치 였던 node 의 right sub tree 로 입양
 - 4. Rule 1 검사

Appendix – Red Black Tree

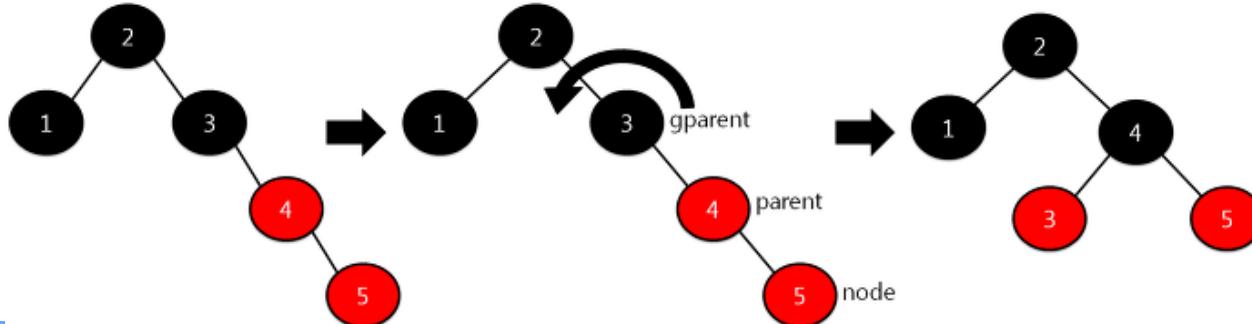
- Algorithm
 - classic red-black tree case by Rue 4 – red node 는 red node 를 자식으로 가질 수 없음
 - 1 ~ 3 insert



- insert 4



- insert 5

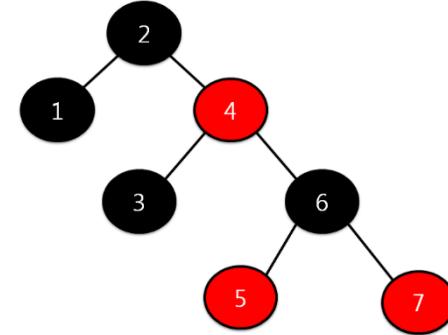
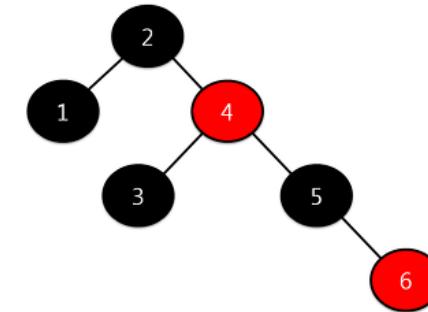
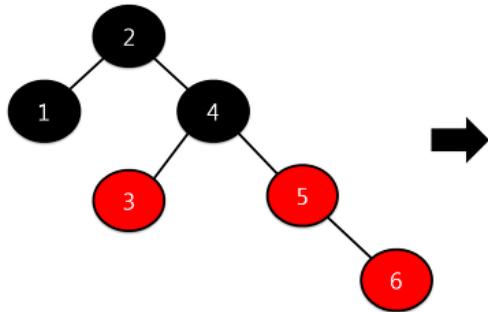


Appendix – Red Black Tree

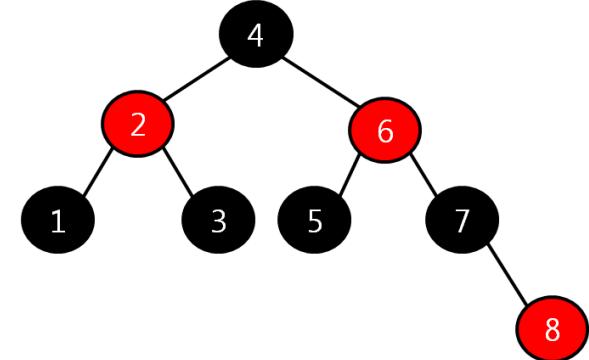
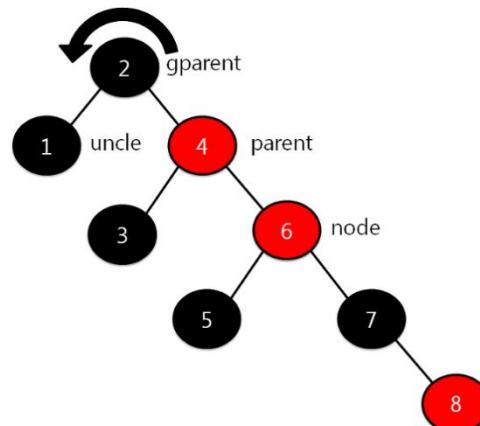
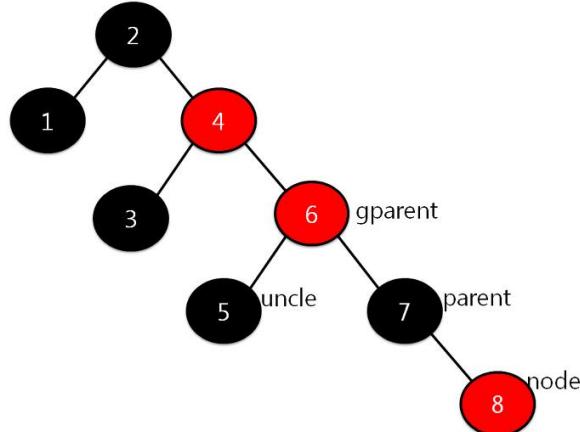
- Algorithm

- classic red-black tree case by Rue 4 – red node 는 red node 를 자식으로 가질 수 없음

- 6,7 insert



- 8 insert



Appendix – Red Black Tree

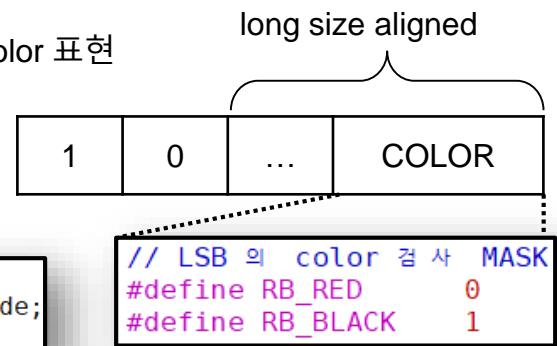
- classic red-black tree kernel API

- rb_node & color

- long 으로 align 되어 있으며 LSB 를 color 으로 사용
 - __rb_parent_color 를 통해 parent node 의 주소 및 현재 node 의 color 표현

```
struct rb_node {
    unsigned long __rb_parent_color;
    // 현재 node 의 색과 부모로의 parent pointer 를 모두 가지는 변수
    // aligne 되어 있기 때문에 LSB 를 color 를 표현하도록 사용하고
    // 나머지를 부모로의 pointer 로 사용
    struct rb_node *rb_right;
    struct rb_node *rb_left;
} __attribute__((aligned(sizeof(long))));
```

```
struct rb_root {
    struct rb_node *rb_node;
};
```



- rb_node macro

- rb_entry : rb_node 를 포함한 구조체 가져옴
 - rb_color : rb_node 의 color 을 검사 및 가져옴
 - rb_next, rb_prev, rb_first, rb_last : iterate 함수

```
#define rb_parent(r) ((struct rb_node *)((r)->__rb_parent_color & ~3))
// rb_node 의 __rb_parent_color 에서 하위 2 bit 가져옴
#define RB_ROOT (struct rb_root) { NULL, }
// rb_root 생성 및 초기화
#define rb_entry(ptr, type, member) container_of(ptr, type, member)
// member 라는 이름으로 rb_node 를 가진 type 형 struct 의 주소를 ptr 에 가져옴
#define RB_EMPTY_ROOT(root) (READ_ONCE((root)->rb_node) == NULL)
// rbtree 가 현재 비어 있는지 검사
/* 'empty' nodes are nodes that are known not to be inserted in an rbtree */
#define RB_EMPTY_NODE(node) \
    ((node)->__rb_parent_color == (unsigned long)(node))
// rb_node 가 현재 rbtree 에 추가되어 있는 상태인지 검사
#define RB_CLEAR_NODE(node) \
    ((node)->__rb_parent_color = (unsigned long)(node))
// rb_node 생성 시, parent 를 가리키는 변수가 자기 자신을
// 가리키도록 초기화
```

```
static inline void rb_set_parent_color(struct rb_node *rb,
                                       struct rb_node *p, int color)
{
    rb->__rb_parent_color = (unsigned long)p | color;
    // 지정된 color 와 parent 주소 설정
}
```

```
#define __rb_color(pc) ((pc) & 1)
#define __rb_is_black(pc) __rb_color(pc)
#define __rb_is_red(pc) (!__rb_color(pc))
#define rb_color(rb) __rb_color((rb)->__rb_parent_color)
// rb_node 의 __rb_parent_color 의 LSB 검사를 통해 color 가져옴
#define rb_is_red(rb) __rb_is_red((rb)->__rb_parent_color)
// rb 의 __rb_parent_color 의 LSB 가 0 인지 검사
#define rb_is_black(rb) __rb_is_black((rb)->__rb_parent_color)
// rb 의 __rb_parent_color 의 LSB 가 1 인지 검사
```



Appendix – Red Black Tree

- classic red-black tree kernel API
 - functions
 - rb_link_node
 - 위치가 결정된 node (leaf node) 를 rb_link 에 연결
 - color 을 그냥 parent 로만 초기화 시킨 -> new node 가 red 이기 때문

```
static inline void rb_link_node(struct rb_node *node, struct rb_node *parent,
                                struct rb_node **rb_link)
{
    node->_rb_parent_color = (unsigned long)parent;
    // 처음 node 를 삽입 할 때 블랙지기 위한 함수로 __rb_parent_color 를
    // parent 로 설정 한다는 것은 node 의 color 를 red 로 설정 한다는
    // 의미이기도 함
    node->rb_left = node->rb_right = NULL;

    *rb_link = node;
    // 이 node 가 추가될 위치를 의미 rb_link_node 함수의 호출 전 left,right 의
    // 위치가 결정되고 결정된 위치의 parent->rb_left 또는 parent->rb_right 를
    // 넘겨 주어 tree 연결 수행
    //
    // 이 함수 호출 후, rb_insert_color 또는 __rb_insert 호출을 통해
    // balance 수행
}
```

- rb_erase
 - 해당 node 를 삭제하며 balancing 수행

```
void rb_erase(struct rb_node *node, struct rb_root *root)
{
    struct rb_node *rebalance;
    rebalance = __rb_erase_augmented(node, root, &dummy_callbacks);
    if (rebalance)
        __rb_erase_color(rebalance, root, dummy_rotate);
}
```

- rb_insert, rb_erase_color
 - rule 을 맞추기 위한 balancing 수행
 - node 삽입, 삭제 후 호출

Appendix – Red Black Tree

- example code from kernel Documentation

- data node

```
struct mytype {  
    struct rb_node node;  
    char *keystring;  
};
```

- search

```
struct mytype *my_search(struct rb_root *root, char *string)  
{  
    struct rb_node *node = root->rb_node;  
  
    while (node) {  
        struct mytype *data = container_of(node, struct mytype, node);  
        int result;  
  
        result = strcmp(string, data->keystring);  
  
        if (result < 0)  
            node = node->rb_left;  
        else if (result > 0)  
            node = node->rb_right;  
        else  
            return data;  
    }  
    return NULL;  
}
```

- erase

```
struct mytype *data = mysearch(&mytree, "walrus");  
  
if (data) {  
    rb_erase(&data->node, &mytree);  
    myfree(data);  
}
```

- insert

```
int my_insert(struct rb_root *root, struct mytype *data)  
{  
    struct rb_node **new = &(root->rb_node), *parent = NULL;  
  
    /* Figure out where to put new node */  
    while (*new) {  
        struct mytype *this = container_of(*new, struct mytype, node);  
        int result = strcmp(data->keystring, this->keystring);  
  
        parent = *new;  
        if (result < 0)  
            new = &((*new)->rb_left);  
        else if (result > 0)  
            new = &((*new)->rb_right);  
        else  
            return FALSE;  
    }  
  
    /* Add new node and rebalance tree. */  
    rb_link_node(&data->node, parent, new);  
    rb_insert_color(&data->node, root);  
  
    return TRUE;  
}
```

Appendix – Augmented Red Black Tree

augmented rbtree & kernel API

- 각 node 마다 추가적인 정보를 활용하여 추가, 검색, 삭제 등의 시간을 줄이기 위한 rbtree 의 확장
- vm_area_struct 관리, Distributed Replicated Block Device 등에서 사용
- rbtree 에 node 가 insert, remove 될 때, balancing 될 때마다 수행되도록 하는 callback 함수 등록
- 주어진 range 에 포함되는 interval 검색, 삭제, 추가 등 수행 이 있을 때마다 callback 함수가 수행되어 추가적인 정보를 update 함으로써 추후 정보를 활용
- augment value 를 update 하는 세가지 함수 제공
 - propagate macro
 - copy
 - rotate
- RB_DECLARE_CALLBACK MACRO 를 통해 augment value 수정 function 선언 및 초기화

```
struct rb_augment_callbacks {
    void (*propagate)(struct rb_node *node, struct rb_node *stop);
    // node 가 stop 이라는 상위 node 에 도달 할 때 까지 augmented value 를
    // update 수행
    void (*copy)(struct rb_node *old, struct rb_node *new);
    // old 의 node 를 root 로 하는 subtree 의 augmented value 를
    // new 의 node 를 root 로 하는 subtree 의 augmented value 에 복사
    void (*rotate)(struct rb_node *old, struct rb_node *new);
    // copy 와 같은 일 하고 난 후 , old 에 대해 augment value 재 계산
};
```

```
#define RB_DECLARE_CALLBACKS(rbstatic, rbname, rbstruct, rbfield, \
                           rbtype, rbaugmented, rbcompute) \
static inline void \
rbname ## _propagate(struct rb_node *rb, struct rb_node *stop) \
{ \
    while (rb != stop) { \
        rbstruct *node = rb_entry(rb, rbstruct, rbfield); \
        rbtype augmented = rbcompute(node); \
        if (node->rbaugmented == augmented) \
            break; \
        node->rbaugmented = augmented; \
        rb = rb_parent(&node->rbfield); \
    } \
} \
static inline void \
rbname ## _copy(struct rb_node *rb_old, struct rb_node *rb_new) \
{ \
    rbstruct *old = rb_entry(rb_old, rbstruct, rbfield); \
    rbstruct *new = rb_entry(rb_new, rbstruct, rbfield); \
    new->rbaugmented = old->rbaugmented; \
} \
static void \
rbname ## _rotate(struct rb_node *rb_old, struct rb_node *rb_new) \
{ \
    rbstruct *old = rb_entry(rb_old, rbstruct, rbfield); \
    rbstruct *new = rb_entry(rb_new, rbstruct, rbfield); \
    new->rbaugmented = old->rbaugmented; \
    old->rbaugmented = rbcompute(old); \
} \
rbstatic const struct rb_augment_callbacks rbname = { \
    .propagate = rbname ## _propagate, \
    .copy = rbname ## _copy, \
    .rotate = rbname ## _rotate \
};
```

Appendix – Augmented Red Black Tree

- augmented rbtree & kernel API
 - __rb_change_child
 - __rb_change_child_cru
 - __rb_erase_augmented
 - remove 및 balance 함수
 - rb_erase_augmented
 - remove 및 balance 함수
 - rb_insert_augmented
 - insert 및 balance
 - rb_set_parent
 - rb_set_parent_color

Appendix – Augmented Red Black Tree

augmented rbtree 를 사용한 mm 에서의 vma 관리

- 각 mm마다 관리하는 vma는 augmented rbtree를 사용하여 관리하며 augmented value로 vma->rb_subtree_gap 사용
- vma->rb_subtree_gap : rb_left, rb_right 각각의 subtree에 존재하는 free virtual address space 크기 중 최대 값

```
#define RB_DECLARE_CALLBACKS(rbstatic, rbname, rbstruct, rbfield, \
    rbtype, rbaugmented, rbcompute) \
static inline void \
rbname ## _propagate(struct rb_node *rb, struct rb_node *stop) \
{ \
    while (rb != stop) { \
        rbstruct *node = rb_entry(rb, rbstruct, rbfield); \
        rbtype augmented = rbcompute(node); \
        if (node->rbaugmented == augmented) \
            break; \
        node->rbaugmented = augmented; \
        rb = rb_parent(&node->rbfield); \
    } \
} \
static inline void \
rbname ## _copy(struct rb_node *rb_old, struct rb_node *rb_new) \
{ \
    rbstruct *old = rb_entry(rb_old, rbstruct, rbfield); \
    rbstruct *new = rb_entry(rb_new, rbstruct, rbfield); \
    new->rbaugmented = old->rbaugmented; \
} \
static void \
rbname ## _rotate(struct rb_node *rb_old, struct rb_node *rb_new) \
{ \
    rbstruct *old = rb_entry(rb_old, rbstruct, rbfield); \
    rbstruct *new = rb_entry(rb_new, rbstruct, rbfield); \
    new->rbaugmented = old->rbaugmented; \
    old->rbaugmented = rbcompute(old); \
} \
rbstatic const struct rb_augment_callbacks rbname = { \
    .propagate = rbname ## _propagate, \
    .copy = rbname ## _copy, \
    .rotate = rbname ## _rotate \
};
```

```
RB_DECLARE_CALLBACKS(static, vma_gap_callbacks, struct vm_area_struct, vm_rb, \
    unsigned long, rb_subtree_gap, vma_compute_subtree_gap) \
static long vma_compute_subtree_gap(struct vm_area_struct *vma) \
{ \
    unsigned long max, subtree_gap; \
    max = vma->vm_start; \
    if (vma->vm_prev) \
        max -= vma->vm_prev->vm_end; \
    // 그 전 vma의 vm_end 와 현재 vma의 vm_start 사이에 \
    // 얼마나큼의 free 가 있는지 구함 \
    if (vma->vm_rb.rb_left) { \
        // left child 가 있다면 left child 의 rb_subtree_gap 을 \
        subtree_gap = rb_entry(vma->vm_rb.rb_left, \
            struct vm_area_struct, vm_rb)->rb_subtree_gap; \
        if (subtree_gap > max) \
            max = subtree_gap; \
        // 현재 vma 값과 비교하여 최대값 구함 \
    } \
    if (vma->vm_rb.rb_right) { \
        // 오른쪽 자식도 마찬가지 작 없 수 행 \
        subtree_gap = rb_entry(vma->vm_rb.rb_right, \
            struct vm_area_struct, vm_rb)->rb_subtree_gap; \
        if (subtree_gap > max) \
            max = subtree_gap; \
    } \
    return max; \
}
```

Appendix – Augmented Red Black Tree

augmented rbtree 를 사용한 mm 에서의 vma 관리

- 각 mm마다 관리하는 vma는 augmented rbtree를 사용하여 관리하며 augmented value로 vma->rb_subtree_gap 사용
- vma->rb_subtree_gap : rb_left, rb_right 각각의 subtree에 존재하는 free virtual address space 크기 중 최대 값

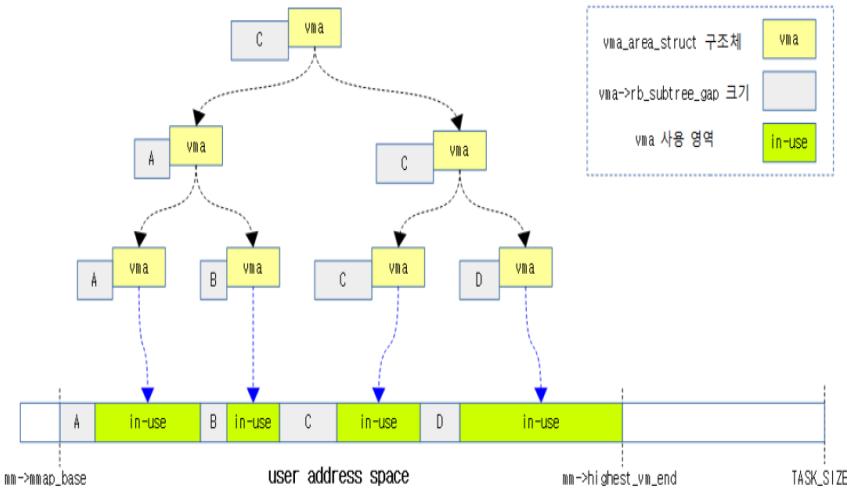
```
#define RB_DECLARE_CALLBACKS(rbstatic, rbname, rbstruct, rbfield, \
    rbtype, rbaugmented, rbcompute) \
static inline void \
rbname ## _propagate(struct rb_node *rb, struct rb_node *stop) \
{ \
    while (rb != stop) { \
        rbstruct *node = rb_entry(rb, rbstruct, rbfield); \
        rbtype augmented = rbcompute(node); \
        if (node->rbaugmented == augmented) \
            break; \
        node->rbaugmented = augmented; \
        rb = rb_parent(&node->rbfield); \
    } \
} \
static inline void \
rbname ## _copy(struct rb_node *rb_old, struct rb_node *rb_new) \
{ \
    rbstruct *old = rb_entry(rb_old, rbstruct, rbfield); \
    rbstruct *new = rb_entry(rb_new, rbstruct, rbfield); \
    new->rbaugmented = old->rbaugmented; \
} \
static void \
rbname ## _rotate(struct rb_node *rb_old, struct rb_node *rb_new) \
{ \
    rbstruct *old = rb_entry(rb_old, rbstruct, rbfield); \
    rbstruct *new = rb_entry(rb_new, rbstruct, rbfield); \
    new->rbaugmented = old->rbaugmented; \
    old->rbaugmented = rbcompute(old); \
} \
rbstatic const struct rb_augment_callbacks rbname = { \
    .propagate = rbname ## _propagate, \
    .copy = rbname ## _copy, \
    .rotate = rbname ## _rotate \
};
```

```
RB_DECLARE_CALLBACKS(static, vma_gap_callbacks, struct vm_area_struct, vm_rb, \
    unsigned long, rb_subtree_gap, vma_compute_subtree_gap) \
static inline void \
vma_gap_callbacks_propagate(struct rb_node *rb, struct rb_node *stop) \
{ \
    while (rb != stop) { \
        // stop node에 도달 할 때 까지 반복 \
        struct vm_area_struct *node = rb_entry(rb, struct vm_area_struct, vm_rb); \
        unsigned long long augmented = vma_compute_subtree_gap(node); \
        // vma의 augmented 값을 계산하여 가져옴 \
        if (node->rbaugmented == augmented) \
            break; \
        // 가져와 계산한 값이 달라진다면 \
        node->rbaugmented = augmented; \
        // 값 update하고 \
        rb = rb_parent(&node->vm_rb); \
        // parent node로 이동하여 반복 수행 \
    } \
} \
static inline void \
vma_gap_callbacks_copy(struct rb_node *rb_old, struct rb_node *rb_new) \
{ \
    struct vm_area_struct *old = rb_entry(rb_old, struct vm_area_struct, vm_rb); \
    struct vm_area_struct *new = rb_entry(rb_new, struct vm_area_struct, vm_rb); \
    new->rbaugmented = old->rbaugmented; \
    // rb_old의 augmented value를 new로 복사 \
} \
static void \
vma_gap_callbacks_rotate(struct rb_node *rb_old, struct rb_node *rb_new) \
{ \
    struct vm_area_struct *old = rb_entry(rb_old, struct vm_area_struct, vm_rb); \
    struct vm_area_struct *new = rb_entry(rb_new, struct vm_area_struct, vm_rb); \
    new->rbaugmented = old->rbaugmented; \
    // 복사하고 \
    old->rbaugmented = vma_compute_subtree_gap(old); \
    // old의 값을 재계산 \
} \
static const struct rb_augment_callbacks vma_gap_callbacks = { \
    .propagate = vma_gap_callbacks_propagate, \
    .copy = vma_gap_callbacks_copy, \
    .rotate = vma_gap_callbacks_rotate \
};
```

Appendix – Augmented Red Black Tree

- augmented rbtree 를 사용한 mm 에서의 vma 관리

- 각 mm마다 관리하는 vma는 augmented rbtree를 사용하여 관리하며 augmented value로 vma->rb_subtree_gap 사용
- vma->rb_subtree_gap : rb_left, rb_right 각각의 subtree에 존재하는 free virtual address space 크기 중 최대 값



```
RB_DECLARE_CALLBACKS(static, vma_gap_callbacks, struct vm_area_struct, vm_rb,
                     unsigned long, rb_subtree_gap, vma_compute_subtree_gap)

static inline void
vma_gap_callbacks_propagate(struct rb_node *rb, struct rb_node *stop)
{
    while (rb != stop) {
        // stop node에 도달 할 때 까지 반복
        struct vm_area_struct *node = rb_entry(rb, struct vm_area_struct, vm_rb);
        unsigned long long augmented = vm_compute_subtree_gap(node);
        // vma의 augmented 값을 계산 하여 가져옴
        if (node->rbaugmented == augmented)
            break;
        // 가져 와 계산 한 값이 달라진다면
        node->rbaugmented = augmented;
        // 값 update 하고
        rb = rb_parent(&node->vm_rb);
        // parent node로 이동하여 반복 수행
    }
}
static inline void
vma_gap_callbacks_copy(struct rb_node *rb_old, struct rb_node *rb_new)
{
    struct vm_area_struct *old = rb_entry(rb_old, struct vm_area_struct, vm_rb);
    struct vm_area_struct *new = rb_entry(rb_new, struct vm_area_struct, vm_rb);
    new->rbaugmented = old->rbaugmented;
    // rb_old의 augmented value 를 new로 복사
}
static void
vma_gap_callbacks_rotate(struct rb_node *rb_old, struct rb_node *rb_new)
{
    struct vm_area_struct *old = rb_entry(rb_old, struct vm_area_struct, vm_rb);
    struct vm_area_struct *new = rb_entry(rb_new, struct vm_area_struct, vm_rb);
    new->rbaugmented = old->rbaugmented;
    // 복사하고
    old->rbaugmented = vm_compute_subtree_gap(old);
    // old의 값을 재계산
}
static const struct rb_augment_callbacks vma_gap_callbacks = {
    .propagate = vma_gap_callbacks_propagate,
    .copy = vma_gap_callbacks_copy,
    .rotate = vma_gap_callbacks_rotate
};
```

Management of Regions

- Representation of Regions

- vma 와 관련된 operation 들을 모아놓은 vm_operations_struct 를 통해 관리

```
struct vm_operations_struct {
    void (*open)(struct vm_area_struct * area);
    // vm_area_struct 가 새로 생성될 때 (e.g. fork 시 ), vma 관련 설정
    void (*close)(struct vm_area_struct * area);
    // vm_area_struct 가 삭제될 때
    int (*mremap)(struct vm_area_struct * area);
    int (*fault)(struct vm_fault *vmf);
    // physical memory 가 없는 page 에 access 시 , page fault handler 가
    // 호출하는 함수
    int (*huge_fault)(struct vm_fault *vmf, enum page_entry_size pe_size);
    void (*map_pages)(struct vm_fault *vmf,
                      pgoff_t start_pgoff, pgoff_t end_pgoff);

    /* notification that a previously read-only page is about to become
     * writable, if an error is returned it will cause a SIGBUS */
    int (*page_mkwrite)(struct vm_fault *vmf);
    // read only page 를 writable 하게 변경 시 , page fault handler 가
    // 호출하는 함수
    /* same as page_mkwrite when using VM_PFNMAP|VM_MIXEDMAP */
    int (*pfn_mkwrite)(struct vm_fault *vmf);

    /* called by access_process_vm when get_user_pages() fails, typically
     * for use by special VMAs that can switch between memory and hardware
     */
    int (*access)(struct vm_area_struct *vma, unsigned long addr,
                  void *buf, int len, int write);
    // get_user_pages 호출 시 , access_process_vm 에서 호출하는 함수
    /* Called by the /proc/PID/maps code to ask the vma whether it
     * has a special name. Returning non-NULL will also cause this
     * vma to be dumped unconditionally. */
    const char *(*name)(struct vm_area_struct *vma);
```

Operations on Regions

Associating Virtual Address with a Region

- address < vm_end 만족하는 첫 vma 검색
- task_struct 별로 vma cache 관리
 - struct vm_area_struct mmap_cache (1 개)
-> struct vmcache (vma 4 개)
 - vmcache 별로 sequence number 를 가지며
vmcache 가 속한 mm 도 이를 가짐

```
/* Look up the first VMA which satisfies addr < vm_end, NULL if none. */
struct vm_area_struct *find_vma(struct mm_struct *mm, unsigned long addr)
{
    // vm_end 항목이 addr 보다 큰 첫 번째 memory area 를 찾는 함수로 addr 이
    // vma 에 포함되어 있는 것을 보장 하지는 않음
    struct rb_node *rb_node;
    struct vm_area_struct *vma;

    /* Check the cache first. */
    vma = vmacache_find(mm, addr);
    // task_struct 별로 가지고 있는
    if (likely(vma))
        return vma;
    // cache에 없으므로 이제 rb tree에서 찾아야 함
    // 일단 mm에서 rbtree의 root node를 가져옴
    rb_node = mm->mm_rb.rb_node;

    while (rb_node) {
        struct vm_area_struct *tmp;

        tmp = rb_entry(rb_node, struct vm_area_struct, vm
        // rb_node에서부터 순회하며 container_of로 해듬
        // vm_area_struct 가져옴
        // ... --vm_end ~ vm_start-----vm_end ~ v
        //           |           |
        // 0 < ... +++++++*++++*++++*++++*++++*+++++
        //           |           |           |
        //         c1)addr     c2)addr     c3)
        if (tmp->vm_end > addr) {
            vma = tmp;
            if (tmp->vm_start <= addr)
                break;
            // case 2 => addr에 해당하는 vma 찾음
            rb_node = rb_node->rb_left;
            // case 1 => left child로 이동
        } else
            rb_node = rb_node->rb_right;
        // case 3 => right child로 이동
    }
}
```

```
struct vm_area_struct *vmacache_find(struct mm_struct *mm, unsigned long addr)
{
    int i;

    count_vm_vmacache_event(VMACACHE_FIND_CALLS);

    if (!vmacache_valid(mm))
        return NULL;
    // current task_struct의 vmacache의 sequence number가 넘겨진
    // mm가 가진 vm_area_struct를 인지 검사 아닐 경우 및 current
    // task_struct의 vmacache 재설정
    for (i = 0; i < VMACACHE_SIZE; i++) {
        struct vm_area_struct *vma = current->vmacache.vmas[i];

        if (!vma)
            continue;
        if (WARN_ON_ONCE(vma->vm_mm != mm))
            break;
        if (vma->vm_start <= addr && vma->vm_end > addr) {
            // addr이 vma의 start ~ end에 위치한다면 해당하는 vma 찾은 것
            count_vm_vmacache_event(VMACACHE_FIND_HITS);
            return vma;
        }
    }
}

return NULL;
```

Operations on Regions

Associating Virtual Address with a Region

- address < vm_end 만족하는 첫 vma 검색
- task_struct 별로 vma cache 관리
 - struct vm_area_struct mmap_cache (1 개)
-> struct vmcache (vma 4 개)
 - vmcache 별로 sequence number 를 가지며
vmcache 가 속한 mm 도 이를 가짐
- addr 에 해당하는 vma 를 찾기 전 먼저
vmcache 를 검증 및 확인

```
/* Look up the first VMA which satisfies addr < vm_end, NULL if none. */
struct vm_area_struct *find_vma(struct mm_struct *mm, unsigned long addr)
{
    // vm_end 항목이 addr 보다 큰 첫 번째 memory area 를 찾는 함수로 addr 이
    // vma 에 포함되어 있는 것을 보장하지는 않음
    struct rb_node *rb_node;
    struct vm_area_struct *vma;

    /* Check the cache first. */
    vma = vmacache_find(mm, addr);
```

```
struct task_struct {
    ...
    /* Per-thread vma caching: */
    struct vmacache vmacache;
    // 최근에 접근된 vm_area_struct 를
    // 가지고 있는 vmacache
    // cache size 는 4 개 일
    ...
} struct vmacache {
    u32 seqnum;
    struct vm_area_struct *vmas[VMACACHE_SIZE];
};

// ... --vm_end ~ vm_start-----vm
//           |
// 0 <-- ... ++++++*++++*++++*+++++
//           |           |
//           c1)addr     c2)addr
```

```
struct mm_struct {
    ...
    u32 vmacache_seqnum;
    /* per-thread vmacache */
    // task_struct 별로 가지고 있는
    // VMCACHE_SIZE 크기 (vm_area_struct 4개)의
    // vmacache 에 해당하는 sequence number
}
...
// case 3 => right child 로 이동
```

```
static bool vmacache_valid(struct mm_struct *mm)
{
    struct task_struct *curr;

    if (!vmacache_valid_mm(mm))
        return false;

    curr = current;
    if (mm->vmacache_seqnum != curr->vmacache.seqnum) {
        /*
         * First attempt will always be invalid, initialize
         * the new cache for this task here.
         */
        // current task_struct 의 vmacache 가 넘겨진 mm 에 속한 것이 아닐 경우 ,
        // current task_struct 의 vmacache 새로 설정 및 cache flush
        curr->vmacache.seqnum = mm->vmacache_seqnum;
        vmacache_flush(curr);
        return false;
    }
    return true;
}

}
return NULL;
```

서 찾아야
를 가져온

Operations on Regions

Associating Virtual Address with a Region

- address < vm_end 만족하는 첫 vma 검색
- task_struct 별로 vma cache 관리
 - struct vm_area_struct mmap_cache (1 개)
-> struct vmcache (vma 4 개)
 - vmcache 별로 sequence number 를 가지며
vmcache 가 속한 mm 도 이를 가짐
- addr 에 해당하는 vma 를 찾기 전 먼저
vmcache 를 검증 및 확인
- cache 유효할 경우 cache 에서 찾아질 수 있는지
address 범위 검사

```
/* Look up the first VMA which satisfies addr < vm_end, NULL if none. */
struct vm_area_struct *find_vma(struct mm_struct *mm, unsigned long addr)
{
    // vm_end 항목이 addr 보다 큰 첫 번째 memory area 를 찾는 함수로 addr 이
    // vma 에 포함되어 있는 것을 보장하지는 않음
    struct rb_node *rb_node;
    struct vm_area_struct *vma;

    /* Check the cache first. */
    vma = vmacache_find(mm, addr);
```

```
struct task_struct {
    ...
    /* Per-thread vma caching: */
    struct vmacache vmacache;
    // 최근에 접근된 vm_area_struct 를
    // 가지고 있는 vmacache
    // cache size 는 4 개 일
}
```

```
} struct vmacache {
    u32 seqnum;
    struct vm_area_struct *vmas[VMACACHE_SIZE]; // 해당
}; // ... --vm_end ~ vm_start-----vm_end ~ v
// | | |
// 0 < ... +++++++*++++*++++*++++*+++++++
// | | |
// c1)addr c2)addr c3)
```

```
struct mm_struct {
    ...
    u32 vmacache_seqnum;
    /* per-thread vmacache */
    // task_struct 별로 가지고 있는
    // VMACACHE_SIZE 크기 (vm_area_struct 4개)의
    // vmacache 에 해당하는 sequence number
}
...
// case 3 => right child 로 이동
```

서 찾아야 할
를 가져 올

```
struct vm_area_struct *vmacache_find(struct mm_struct *mm, unsigned long addr)
{
    int i;
    count_vm_vmacache_event(VMACACHE_FIND_CALLS);

    if (!vmacache_valid(mm))
        return NULL;
    // current task_struct 의 vmacache 의 sequence number 가 넘겨진
    // mm 가 가진 vm_area_struct 를 인지 검사 아닐 경우 및 current
    // task_struct 의 vmacache 재설정
    for (i = 0; i < VMACACHE_SIZE; i++) {
        struct vm_area_struct *vma = current->vmacache.vmas[i];

        if (!vma)
            continue;
        if (WARN_ON_ONCE(vma->vm_mm != mm))
            break;
        if (vma->vm_start <= addr && vma->vm_end > addr) {
            // addr 이 vma 의 start ~ end 에 위치한다면 해당하는 vma 찾은 것
            count_vm_vmacache_event(VMACACHE_FIND_HITS);
            return vma;
        }
    }
    return NULL;
}
```

Operations on Regions

Associating Virtual Address with a Region

- task_struct 별로 vma cache 관리
 - struct vm_area_struct mmap_cache (1 개)
-> struct vmcache (vma 4 개)
 - vmcache 별로 sequence number 를 가지며
vmcache 가 속한 mm 도 이를 가짐
- addr 에 해당하는 vma 를 찾기 전 먼저
vmcache 를 검증 및 확인
- cache 유효할 경우 cache 에서 찾아질 수 있는지
address 범위 검사
- cache 에서 찾았지 않을 경우 rbtree 의 node 를 순회
하며 address range 검사 수행
 - case 2 에 해당 시 vma 찾음
 - case 1 에 해당 시 왼쪽 자식으로 이동
 - case 3 의 경우 오른쪽 자식으로 이동

```
/* Look up the first VMA which satisfies addr < vm_end, NULL if none. */
struct vm_area_struct *find_vma(struct mm_struct *mm, unsigned long addr)
{
    // vm_end 항목이 addr 보다 큰 첫 번째 memory area 를 찾는 함수로 addr 이
    // vma 에 포함되어 있는 것을 보장 하지는 않음
    struct rb_node *rb_node;
    struct vm_area_struct *vma;

    /* Check the cache first. */
    vma = vmacache_find(mm, addr);
    // task_struct 별로 가지고 있는
    if (likely(vma))
        return vma;
    // cache 에 없으므로 이제 rb tree 에서 찾아야 함
    // 일단 mm에서 rbtree 의 root node 를 가져옴
    rb_node = mm->mm_rb.rb_node;

    while (rb_node) {
        struct vm_area_struct *tmp;

        tmp = rb_entry(rb_node, struct vm_area_struct, vm_rb);
        //
        // rb_node에서부터 순회하며 container_of로 해당하는
        // vm_area_struct 가져옴
        // ...      --vm_end ~ vm_start-----vm_end ~ vm_start--
        //           |           |           |
        // 0 <- ... +++++++*+++++*++++*++++++*+++++-----> 3G
        //           |           |           |
        //         c1)addr     c2)addr     c3)addr
        //
        if (tmp->vm_end > addr) {
            vma = tmp;
            if (tmp->vm_start <= addr)
                break;
            // case 2 => addr에 해당하는 vma 찾음
            rb_node = rb_node->rb_left;
            // case 1 => left child로 이동
        } else
            rb_node = rb_node->rb_right;
            // case 3 => right child로 이동
    }
}
```

Operations on Regions

```
// r1) exist
// address 정 확 히 포 함 하 는 vma 있 음
//
//      vm_end ~ vm_start --- addr --- vm_end ~ vm_start
//      |           |    변 수 vma   |           |
//      -----vm-----          -----vm-----          -----
//
// r2) not exist
// address 를 포 함 하 는 vma 없 고 address 가 다른 vma 사 이 에
// 있 음
//      *
//      vm_start --- vm_end ~ addr ~ vm_start --- vm_end ~ vm_start
//      |           |           |    변 수 vma   |           |
//      -----vm-----          -----vm-----          -----
//
// r3) not exist (vma : null)
// address 가 vma 들 의 맨 끝에 있 음 즉 현재 할당 된 vma 들 의
// vma_end 를 과 비교 하 여 제 일 큰 위치
//      *
//      vm_start --- vm_end ~ addr
//      |           |
//      -----vm-----
//
// r4) not exist (vma: null)
// 처음 vma 없 는 상 태 일
//      *
//      addr
//
// 검색 결과 vaddress 가 vma 들 사 이 에 있 어
//
if (vma)
    vmacache_update(addr, vma);
// 찾 은 vma 를 task_struct 의 vmacache 에 추加
return vma;
}
```

Associating Virtual Address with a Region

- task_struct 별로 vma cache 관리
 - struct vm_area_struct mmap_cache (1 개)
-> struct vmcache (vma 4 개)
 - vmcache 별로 sequence number 를 가지며
vmcache 가 속한 mm 도 이를 가짐
- addr 에 해당하는 vma 를 찾기 전 먼저
vmcache 를 검증 및 확인
- cache 유효할 경우 cache 에서 찾아질 수 있는지
address 범위 검사
- cache 에서 찾았지 않을 경우 rbtree 의 node 를 순회
하며 address range 검사 수행
 - case 2 에 해당 시 vma 찾음
 - case 1 에 해당 시 왼쪽 자식으로 이동
 - case 3 의 경우 오른쪽 자식으로 이동

Operations on Regions

- Associating Regions

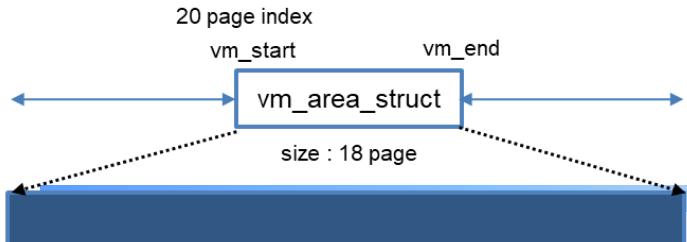
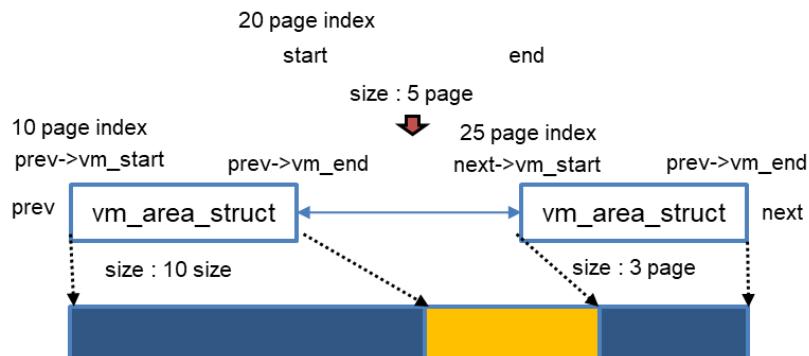
- find_vma_intersection 을 통해 address < vm_end 조건이 아닌 start address ~ end address 와 중첩되는 vm_start ~ vm_end 를 가진 vma 를 반환

```
static inline struct vm_area_struct * find_vma_intersection(struct mm_struct * mm,
    unsigned long start_addr, unsigned long end_addr)
{
    // rbtree search 하기 위한 함수
    // find_vma 를 기반으로 동작하지만, 주어진 start_addr ~ end_addr 의
    // 범위와 중첩되는 첫번째 vma 를 반환
    struct vm_area_struct * vma = find_vma(mm,start_addr);
    // start_addr 보다 큰 vm_end 를 가진 첫번째 vm_area_struct 를 반환
    // vm != NULL 일 경우 아래와 같은 상황 가능
    //
    // ... vm_end 1)vm_start      2)vm_start      3)vm_start      vm_end
    //           |                  |                  |                  |
    // ... +++++++*++++*++++++*++++*++++*++++*++++*++++*++++*+
    //           |                  |                  |
    //           start_addr        end_addr
    //
    // ... vm_end 4)vm_start  5)vm_start      vm_end
    //           |                  |                  |
    // ... +++++++*++++*++++*++++++*++++*++++
    //           |                  |
    //           start_addr        end_addr
    //
    if (vma && end_addr <= vma->vm_start)
        vma = NULL;
    // 3번에 해당하면 안겹치므로 null 반환
    return vma;
}
```

Operating on Regions

Merging Regions

- 새로운 region 추가시 다른 region들과 merge 될 수 있는지 확인하여 가능하다면 하나의 sequential region으로 구성
- 대부분 anon region (e.g. heap or stack)에 대해 수행됨
- vma_merge
 - 앞 vm_area_struct 와 합쳐질 수 있는지 검사
 - 앞 vma 와 address가 연결되는지 검사
 - 같은 memory policy를 가지는지 검사
 - can_vma_merge_after
 - filebacked -> 같은 file 인지
 - anonymous -> 같은 anon_vma 속하는지
 - 뒷 vm_area_struct 와 합쳐질 수 있는지 검사
 - vma_adjust**



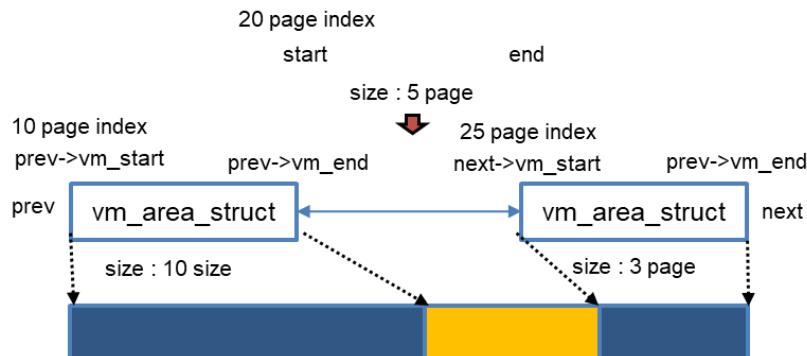
```
struct vm_area_struct *vma_merge(struct mm_struct *mm,
    struct vm_area_struct *prev, unsigned long addr,
    unsigned long end, unsigned long vm_flags,
    struct anon_vma *anon_vma, struct file *file,
    pgoff_t pgoff, struct mempolicy *policy,
    struct vm_userfaultfd_ctx vm_userfaultfd_ctx)

{
    ...
    if (prev && prev->vm_end == addr &&
        mpol_equal(vma_policy(prev), policy) &&
        can_vma_merge_after(prev, vm_flags,
            anon_vma, file, pgoff,
            vm_userfaultfd_ctx)) {
        // 1. 전 vm_area_struct 의 VM 끝 주소와 현재 request 시작
        // 주소가 같은지 검사
        // 2. mempolicy 가 같은지 검사
        // 3. can_vma_merge_after
        //     file backed page라면 같은 file 인지, 같은 속성의 vma 인지
        //     pg_off 기준으로 연속되는지 등을 검사
        /*
         * OK, it can. Can we now merge in the successor as well?
         */
        if (next && end == next->vm_start &&
            mpol_equal(policy, vma_policy(next)) &&
            can_vma_merge_before(next, vm_flags,
                anon_vma, file,
                pgoff+pglen,
                vm_userfaultfd_ctx) &&
            is_mergeable_anon_vma(prev->anon_vma,
                next->anon_vma, NULL)) {
            // 1. 현재 요청 vma 범위의 end가 다음 vma의 start와 같은지 검사
            // 2. mempolicy 가 같은지 검사
            // 3. can_vma_merge_before
            //     위와 같은 검사
            // 4. prev 와 next 도 서로 merge 되어도 되는지 검사
            ...
            /* cases 1, 6 */
            // 앞 뒤 VM 이 모두 merge 가 가능한 상황
            err = __vma_adjust(prev, prev->vm_start,
                next->vm_end, prev->vm_pgoff, NULL,
                prev);
        } else
            /* cases 2, 5, 7 */
            // 앞의 VM 만 merge 가 가능한 상황
            err = __vma_adjust(prev, prev->vm_start,
                end, prev->vm_pgoff, NULL, prev);
        if (err)
            return NULL;
        khugepaged_enter_vma_merge(prev, vm_flags);
        return prev;
    }
    ...
}
```

Operating on Regions

Merging Regions

- 새로운 region 추가시 다른 region들과 merge 될 수 있는지 확인하여 가능하다면 하나의 sequential region으로 구성
- 대부분 anon region (e.g. heap or stack)에 대해 수행됨
- vma_merge
 - 앞 vm_area_struct 와 합쳐질 수 있는지 검사
 - 앞 vma 와 address 가 연결되는지 검사
 - 같은 memory policy 를 가지는지 검사
 - can_vma_merge_after
 - filebacked -> 같은 file 인지
 - anonymous -> 같은 anon_vma 속 하는지
 - 뒷 vm_area_struct 와 합쳐질 수 있는지 검사
 - vma_adjust**



손주형

```

struct vm_area_struct *vma_merge(struct mm_struct *mm,
                                struct vm_area_struct *prev, unsigned long addr,
                                unsigned long end, unsigned long vm_flags,
                                struct anon_vma *anon_vma, struct file *file,
                                struct vm_userfaultfd_ctx vm_userfaultfd_ctx)
{
    if (is_mergeable_vma(vma, file, vm_flags, vm_userfaultfd_ctx) &&
        is_mergeable_anon_vma(anon_vma, vma->anon_vma, vma)) {
        // file backed 라면 같은 파일인지, anon 이라면 같은 anon_vma 인지
        pgoff_t vm_pgoff;
        vm_pglen = vma_pages(vma);
        if (vma->vm_pgoff + vm_pglen == vm_pgoff)
            return 1;
        // vma->vm_pgoff : 해당 vma 의 page 단위 offset
        // vm_pglen : vma 의 page 개수
        // 즉 prior vma 의 page offset 위치부터 그 전 prior vma 의 page 개수
        // 를 더한 값이 새로 추가될 vma 의 page offset 이 되어야 한다.
        // (virtual address 가 연속적이어야 한다.)
    }
    return 0;
}

```

```

static int
can_vma_merge_after(struct vm_area_struct *vma, unsigned long vm_flags,
                    struct anon_vma *anon_vma, struct file *file,
                    pgoff_t vm_pgoff,
                    struct vm_userfaultfd_ctx vm_userfaultfd_ctx)
{
    if (is_mergeable_vma(vma, file, vm_flags, vm_userfaultfd_ctx) &&
        is_mergeable_anon_vma(prev->anon_vma,
                              next->anon_vma, NULL)) {
        static int
can_vma_merge_before(struct vm_area_struct *vma, unsigned long vm_flags,
                     struct anon_vma *anon_vma, struct file *file,
                     pgoff_t vm_pgoff,
                     struct vm_userfaultfd_ctx vm_userfaultfd_ctx)
{
    if (is_mergeable_vma(vma, file, vm_flags, vm_userfaultfd_ctx) &&
        is_mergeable_anon_vma(anon_vma, vma->anon_vma, vma)) {
        // can_vma_merge_after 와 같은 검사
        if (vma->vm_pgoff == vm_pgoff)
            return 1;
        // vm_pgoff 즉 현재 생성하려는 vma 의 pg_off 에 page 개수를 더한
        // 값이 다음 vma 의 vm_pgoff 와 같은지 즉 연속적인지 검사
    }
    return 0;
}
...
}

```

Operating on Regions

● Inserting Regions

○ insert_vm_struct

- find_vma_links (find_vma_prepare에서 바뀜)
 - mm의 rbtree에서 vma가 삽입될 위치를 찾음
- vma_link로 찾은 rbtree 위치에 vma를 넣어 줌

○ vm_link

```
int insert_vm_struct(struct mm_struct *mm, struct vm_area_struct *vma)
{
    struct vm_area_struct *prev;
    struct rb_node **rb_link, *rb_parent;

    if (find_vma_links(mm, vma->vm_start, vma->vm_end,
                       &prev, &rb_link, &rb_parent))
        return -ENOMEM;
    // find_vma_links를 통해 vma를 삽입할 mm에서의 위치를 구함
    // rb_parent : rb_node에서의 부모
    // rb_link : rb_parent에서 연결될 위치

    if ((vma->vm_flags & VM_ACCOUNT) &&
        security_vm_enough_memory_mm(mm, vma_pages(vma)))
        return -ENOMEM;

    /*
     * The vm_pgoff of a purely anonymous vma should be irrelevant
     * until its first write fault, when page's anon_vma and index
     * are set. But now set the vm_pgoff it will almost certainly
     * end up with (unless mremap moves it elsewhere before that
     * first wfault), so /proc/pid/maps tells a consistent story.
     *
     * By setting it to reflect the virtual start address of the
     * vma, merges and splits can happen in a seamless way, just
     * using the existing file pgoff checks and manipulations.
     * Similarly in do_mmap_pgoff and in do_brk.
     */
    if (vma_is_anonymous(vma)) {
        BUG_ON(vma->anon_vma);
        vma->vm_pgoff = vma->vm_start >> PAGE_SHIFT;
    }

    vma_link(mm, vma, prev, rb_link, rb_parent);
    return 0;
}
```

Operating on Regions

Inserting Regions

insert_vm_struct

- find_vma_links (find_vma_prepare 에서 바뀜)
 - mm 의 rbtree 에서 vma 가 삽입 될 위치를 찾음
- vma_link 로 찾은 rbtree 위치에 vma 를 넣어 줌

vm_link

```
static int find_vma_links(struct mm_struct *mm, unsigned long addr,
    unsigned long end, struct vm_area_struct **pprev,
    struct rb_node ***rb_link, struct rb_node **rb_parent)
{
    struct rb_node **__rb_link, *_rb_parent, *rb_prev;

    __rb_link = &mm->mm_rb.rb_node;
    rb_prev = __rb_parent = NULL;
    // root node 부터 시작하여 rb tree 검색
    while (*__rb_link) {
        // root rbnode 부터 하여 검색 시작
        struct vm_area_struct *vma_tmp;

        __rb_parent = *__rb_link;
        vma_tmp = rb_entry(__rb_parent, struct vm_area_struct, vm_rb);
        // start_addr 보다 큰 vm_end 를 가진 첫 번째 vm_area_struct 를 반환
        // 아래와 같은 상황 가능
        // ...   vm_end~ c1)vm_start  c2)vm_start  c3)vm_start  vm_end ~vm_start
        // 0 <- ... ++++++*++++*++++*++++*++++*++++*++++*+ -> 3G
        //                                |           |           |
        //                                addr       end
        //
        // ...vm_end~ c4)vm_start  c5)vm_start  vm_end
        // 0 <- ... ++++++*++++*++++*++++*++++*++++*+ -> 3G
        //                                |           |           |
        //                                addr       end
        //
        // ...   vm_end~ c6)vm_start      vm_end
        // 0<- ... ++++++*++++*++++*++++*++++*++++*+ -> 3G
        //                                |           |           |
        //                                addr       end
        // __rb_link 는 __rb_parent 에서 vma 가 추가될 위치 즉 left or right 를 의미
        if (vma_tmp->vm_end > addr) {
            /* Fail if an existing vma overlaps the area */
            if (vma_tmp->vm_start < end)
                return -ENOMEM; // case 1, 2, 4, 5 => 겹치는 상황
            __rb_link = &__rb_parent->rb_left; // case 3 => left child로 이동
        } else {
            rb_prev = __rb_parent; // case 6 => right child로 이동
            // rb_prev 를 이동 전 left parent 로 설정
            __rb_link = &__rb_parent->rb_right;
        }
    }
    *pprev = NULL;
    if (rb_prev)
        *pprev = rb_entry(rb_prev, struct vm_area_struct, vm_rb);
    // rb_prev NULL 이면 처음 위치 (그림의 맨 왼쪽 처음 자리)
    *rb_link = __rb_link;
    *rb_parent = __rb_parent; // rb_parent 가 오른쪽 vm
    return 0;
}
```

Operating on Regions

Inserting Regions

insert_vm_struct

- find_vma_links (find_vma_prepare 에서 바뀜)
 - mm 의 rbtree 에서 vma 가 삽입 될 위치를 찾음
- vma_link 로 찾은 rbtree 위치에 vma 를 넣어 줌

vm_link

- __vma_link
 - linked list 에 연결
- __vma_link_file
 - rb tree 에 추가
- mm 의 vma 개수 증가

```
static void vma_link(struct mm_struct *mm, struct vm_area_struct *vma,
                     struct vm_area_struct *prev, struct rb_node **rb_link,
                     struct rb_node *rb_parent)
{
    struct address_space *mapping = NULL;
    // prev - vma - container_of(rb_parent) : vma 아직 연결 안 됨
    if (vma->vm_file) {
        // file backed page 일 경우 먼저 rwsem 잡음
        mapping = vma->vm_file->f_mapping;
        i_mmap_lock_write(mapping);
    }

    vma_link(mm, vma, prev, rb_link, rb_parent);
    // vma 와 prev.container_of(rb_parent) 서로 list 연결 및 rbtree 추가
    vma_link_file(vma);
    // file-backed page 일 경우 vma 를 address_space 에 추가
    if (mapping)
        i_mmap_unlock_write(mapping);

    mm->map_count++;
    validate_mm(mm);
}
```

```
static void
__vma_link(struct mm_struct *mm, struct vm_area_struct *vma,
           struct vm_area_struct *prev, struct rb_node **rb_link,
           struct rb_node *rb_parent)
{
    // prev - vma - container_of(rb_parent) : vma 아직 연결 안 됨
    __vma_link_list(mm, vma, prev, rb_parent);
    // vma 를 prev ~ container_of(rb_parent) 와 서로 연결
    __vma_link_rb(mm, vma, rb_link, rb_parent);
    // vma 를 rb tree 에 추가
    // rb_link : rb_parent 에서 vma->vm_rb 가 추가 될 위치
}
```

Operating on Regions

Inserting Regions

insert_vm_struct

- find_vma_links (find_vma_prepare에서 바뀜)
 - mm의 rbtree에서 vma가 삽입될 위치를 찾음
- vma_link로 찾은 rbtree 위치에 vma를 넣어 줌

vm_link

- __vma_link
 - linked list에 연결
- __vma_link_file
 - rb tree에 추가
- mm의 vma 개수 증가

```
void __vma_link_list(struct mm_struct *mm, struct vm_area_struct *vma,
                      struct vm_area_struct *prev, struct rb_node *rb_parent)
{
    struct vm_area_struct *next;
    vma->vm_prev = prev;
    // vma에서 prev 연결
    if (prev) {
        next = prev->vm_next;
        prev->vm_next = vma;
        // prev에서 vma 연결
    } else {
        // prev가 없다면 맨 처음의 것이므로 mm->mmap을 새로 설정
        mm->mmap = vma;
        if (rb_parent)
            next = rb_entry(rb_parent,
                            struct vm_area_struct, vm_rb);
        else
            next = NULL;
    }
    vma->vm_next = next;
    // vma에서 next로 연결 설정
    if (next)
        next->vm_prev = vma;
    // next에서 vma로 연결 설정
}
```

```
static void vma_link(struct mm_struct *mm, struct vm_area_struct *vma,
                      struct vm_area_struct *prev, struct rb_node **rb_link,
                      struct rb_node *rb_parent)
{
    struct address_space *mapping = NULL;
    // prev - vma - container_of(rb_parent) : vma 아직 연결 안 됨
    if (vma->vm_file) {
        // file backed page 일 경우 먼저 rwsem 잡음
        mapping = vma->vm_file->f_mapping;
        i_mmap_lock_write(mapping);
    }

    vma_link(mm, vma, prev, rb_link, rb_parent);
    // vma와 prev.container_of(rb_parent) 서로 list 연결 및 rbtree 추가
    vma_link_file(vma);
}

void __vma_link_rb(struct mm_struct *mm, struct vm_area_struct *vma,
                   struct rb_node **rb_link, struct rb_node *rb_parent)
{
    /* Update tracking information for the gap following the new vma. */
    if (vma->vm_next)
        vma_gap_update(vma->vm_next);
    // parent node로 올라가며 가장 큰 gap 크기 update
    else
        mm->highest_vm_end = vma->vm_end;
    // next가 없다면 즉 vma가 마지막이라면 mm의 마지막 vm 주소 update

    /*
     * vma->vm_prev wasn't known when we followed the rbtree to find the
     * correct insertion point for that vma. As a result, we could not
     * update the vma vm_rb parents rb_subtree_gap values on the way down.
     * So, we first insert the vma with a zero rb_subtree_gap value
     * (to be consistent with what we did on the way down), and then
     * immediately update the gap to the correct value. Finally we
     * rebalance the rbtree after all augmented values have been set.
     */
    rb_link_node(&vma->vm_rb, rb_parent, rb_link);
    // 현재 vma의 vma->vm_rb를 rb_parent의 rb_link 위치(left/right)에
    // 연결하고 vma->vm_rb의 parent pointer에 rb_parent 설정
    vma->rb_subtree_gap = 0;
    vma_gap_update(vma);
    vma_rb_insert(vma, &mm->mm_rb);
    // balancing
}
```

Operating on Regions

Inserting Regions

insert_vm_struct

- find_vma_links (find_vma_prepare에서 바뀜)
 - mm의 rbtree에서 vma가 삽입될 위치를 찾음
- vma_link로 찾은 rbtree 위치에 vma를 넣어 줌

vm_link

- __vma_link
 - linked list에 연결
- __vma_link_file
 - rb tree에 추가
- mm의 vma 개수 증가

```
void __vma_link_list(struct mm_struct *mm, struct vm_area_struct *vma,
                      struct vm_area_struct *prev, struct rb_node *rb_parent)
{
    struct vm_area_struct *next;
    vma->vm_prev = prev;
    // vma에서 prev 연결
    if (prev) {
        next = prev->vm_next;
        prev->vm_next = vma;
        // prev에서 vma 연결
    } else {
        // prev가 없다면 맨 처음의 것이므로 mm->mmap을 새로 설정
        mm->mmap = vma;
        if (rb_parent)
            next = rb_entry(rb_parent,
                            struct vm_area_struct, vm_rb);
        else
            next = NULL;
    }
    vma->vm_next = next;
    // vma에서 next로 연결 설정
    if (next)
        next->vm_prev = vma;
    // next에서 vma로 연결 설정
}
```

```
static void vma_link(struct mm_struct *mm, struct vm_area_struct *vma,
                      struct vm_area_struct *prev, struct rb_node **rb_link,
                      struct rb_node *rb_parent)
{
    struct address_space *mapping = NULL;
    // prev - vma - container_of(rb_parent) : vma 아직 연결 안 됨
    if (vma->vm_file) {
        // file backed page 일 경우 먼저 rwsem 잡음
        mapping = vma->vm_file->f_mapping;
        i_mmap_lock_write(mapping);
    }

    vma_link(mm, vma, prev, rb_link, rb_parent);
    // vma와 prev.container_of(rb_parent) 서로 list 연결 및 rbtree 추가
    vma_link_file(vma);
}

void __vma_link_rb(struct mm_struct *mm, struct vm_area_struct *vma,
                   struct rb_node **rb_link, struct rb_node *rb_parent)
{
    /* Update tracking information for the gap following the new vma. */
    if (vma->vm_next)
        vma_gap_update(vma->vm_next);
    // parent node로 올라가며 가장 큰 gap 크기 update
    else
        mm->highest_vm_end = vma->vm_end;
    // next가 없다면 즉 vma가 마지막이라면 mm의 마지막 vm 주소 update

    /*
     * vma->vm_prev wasn't known when we followed the rbtree to find the
     * correct insertion point for that vma. As a result, we could not
     * update the vma vm_rb parents rb_subtree_gap values on the way down.
     * So, we first insert the vma with a zero rb_subtree_gap value
     * (to be consistent with what we did on the way down), and then
     * immediately update the gap to the correct value. Finally we
     * rebalance the rbtree after all augmented values have been set.
     */

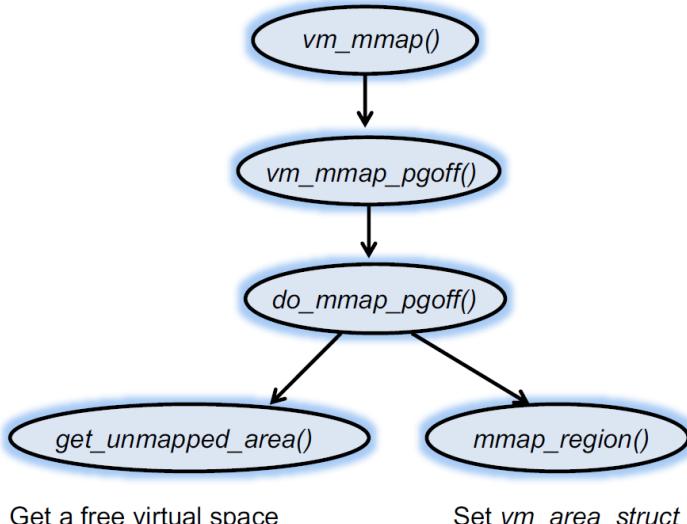
static inline void vma_rb_insert(struct vm_area_struct *vma,
                                 struct rb_root *root)
{
    /* All rb_subtree_gap values must be consistent prior to insertion */
    validate_mm_rb(root, NULL);

    rb_insert_augmented(&vma->vm_rb, root, &vma_gap_callbacks);
}
```

Operating on Regions

Creating Regions

- mmap system call 등의 호출 시 vm_area_struct 생성을 위해 free virtual address range 를 찾아야 함.
- address 가 명시 되어 있을 시, find_vma 를 통해 먼저 해당 주소에 이미 vma 가 있는지 검사 후 없다면 해당 주소 반환
- 없다면 augmented rbtree 를 통해 추가 정보를 가진 gap 을 통해 free virtual address space 를 검색
 - unmap area
 - unmap area topdown



Get a free virtual space

Set vm_area_struct

```
unsigned long
arch_get_unmapped_area(struct file *filp, unsigned long addr,
                      unsigned long len, unsigned long pgoff, unsigned long flags)
{
    struct mm_struct *mm = current->mm;
    struct vm_area_struct *vma;
    struct vm_unmapped_area_info info;

    if (len > TASK_SIZE - mmap_min_addr)
        return -ENOMEM;
    // 요청된 크기가 거의 워 3G 만큼 큰 값인지 검사
    if (flags & MAP_FIXED)
        return addr;
    // addr에 꼭 vma 생성해야 하는지 검사 그럴다면 그냥 반환
    // MAP_FIXED 는 mmap 함수에서 flag로 주어 메모리 시작 번지
    // 지정 하려 할 때 사용
    if (addr) {
        // 주소가 명시되어 있다면 그 주소에 기존 vma 와 겹치는지 확인 해야 함
        addr = PAGE_ALIGN(addr);
        vma = find_vma(mm, addr);
        // addr < vma->vm_end 를 만족하는 vma 가 있는지 찾음
        if (TASK_SIZE - len >= addr && addr >= mmap_min_addr &&
            (!vma || addr + len <= vma->vm_start))
            return addr;
        // addr+len 값이 즉 할당 할 vma 의 끝 주소가 TASK_SIZE 보다 작아야 함
        // addr 위치가 minimum 보다 커야 함
        // find_vma 를 통해 찾은 다음 위치 vma 에 대해 vma 가 null 이어서 뒤에
        // 아무것도 없으므로 할당 가능 한 상태 이거나 , 뒤에 vma 가 있다면
        // vma->start 가 addr+len 보다 커야 함 즉 겹치면 안 됨
    }
    // 주소가 명시되어 있지 않거나 주소에 이미 vma 가 있다면 다른 virtual
    // address 를 찾아야 함
    info.flags = 0;
    info.length = len;
    info.low_limit = mm->mmap_base;
    info.high_limit = TASK_SIZE;
    info.align_mask = 0;
    return vm_unmapped_area(&info);
}
```

Memory Mapping

- mmap, mmap2 system call 을 통해 free virtual address space 를 찾아 vm_area_struct 를 생성
- Creating Mappings
 - sys_mmap -> do_mmap_pgoff -> **do_mmap**
 - get_unmapped_are 를 통해 사용 가능한 free virtual address 영역 가져옴
- Removing Mappings
 - **do_munmap**

Memory Mapping

Nonlinear Mappings

- 하나의 파일 내 여러 부분이 mapping 될 경우, 다수의 vm_area_struct 구성으로 memory cost 를 방지하기 위한 기법
- several pieces of a file into different parts of memory.
- sys_remap_files_page 라는 systemcall 이 있음
 - 하지만 현재는 사용 안함, ABI 를 위해 남겨놓음
 - 현재는 그냥 vma 또 만들어 구성
 - 지금은 memory 단점은 없지만 vma 가 DEFAULT_MAX_MAP_COUNT 를 넘을 가능성이 좀더 높은 단점은 있음)
- 물리 page address 가 들어갈 normal PTE 와 file 내의 offset 이 들어갈 PTE 를 구분하기 위해 PTE flag 사용
 - PTE_FILE_MAX_BITS

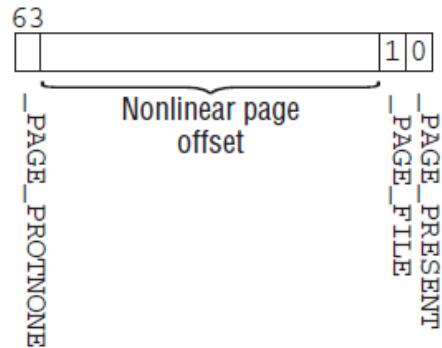


Figure 4-14: Representing nonlinear mappings in page table entries on IA-64 systems.

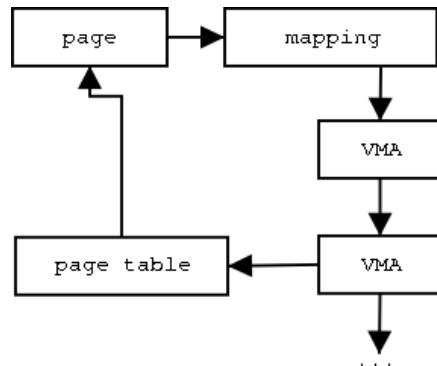
Reverse Mapping

- Data Structures
 - swap, page out 시 해당 page 가 out 되었다는 것을 해당 page 가 사용하는 모든 process 들의 pte 들에 update 해야 하기 때문에 physical page 와 page 가 속한 process 간의 mapping 을 구성하여 관리한다.
 - 해당 page 를 사용하는 process 들의 pate table entry 간의 mapping
 - fork 된 같은 vma 를 관리하기 위해 list 를 만들어 관리한다.
 - 책의 v2.6.39 에서는 file backed page 의 경우 object based reverse mapping (priority based tree) 가 사용되었지만, v4.11 확인 결과 interval red black tree 라는 data structure 사용
 - interval tree vs prio tree
 - insert/erase : 25% faster
 - search : 2.4~3x faster

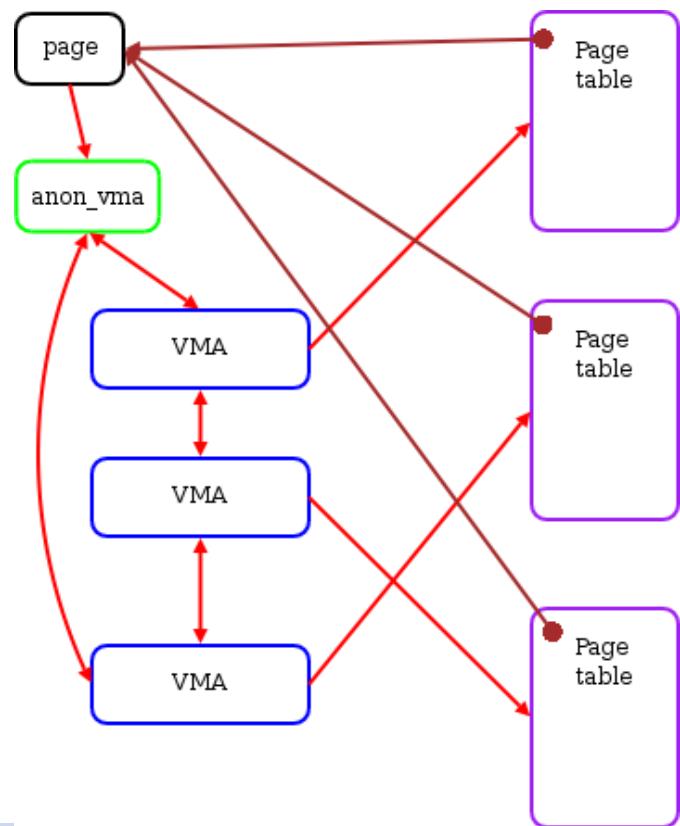
Reverse Mapping

- Reverse mapping history

- Step 1. swapping, page out 발생 시마다, 모든 page table 들을 scan 함(v2.4)
 - Step 2. page struct에 해당 mapping을 가진 여러 process들의 page table에 대한 list_head를 가지도록 함(v2.5)
 - memory 사용량 너무 많음
 - fork 시 속도 느려짐
 - Step 3. page struct에 해당 pte list 삭제하고, address_space 내의 vm_area_struct를 통해 pte 접근
 - pte direct pointer보다 long way지만, pte direct map 지움으로써 lowmem 사용량 줄임
 - file-backed에만 해당



- Step 4. anonymous page 에도 적용하기 위해 anon_vma struct 를 생성하고 해당 anonymous page frame 와 관련된 vma 연결시킴
 - e.g. fork 되면 child 의 vma 들이 모두 parent 의 page->mapping 인 anon_vma 의 list 에 포함됨

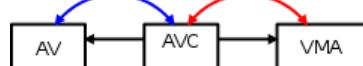


Reverse Mapping

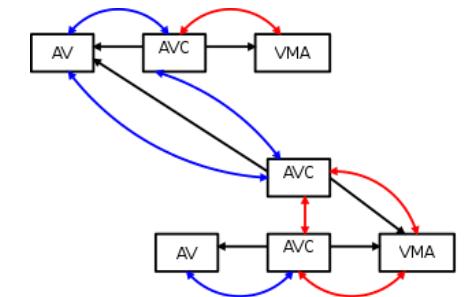
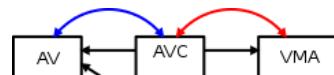
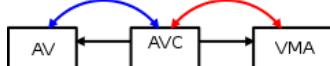
- Reverse mapping history

- Step 5

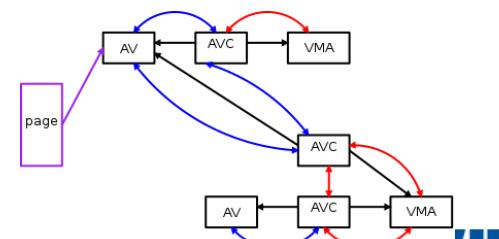
- 1K 개의 child process 를 가지며, 1K 개의 vma 를 가진다면... 하나의 parent process 에 1M 개의 vma 가 list 로 연결되며 rmap 시 1M 번 iterate 해야 함(복사된 해당 page 관련 vma 뿐만 아니라 다른 page 관련 vma 까지 다 iterate)
 - process 마다 anon_vma 를 가지도록 하며, vma 가 연결되는 대신 anon_vma 가 anon_vma_chain 을 통해 연결
 - 구조
 - 각각의 page 는 아래와 같은 구조를 가지게 됨 page->mapping 가 anon_vma 를 가리키고, anon_vma 는 vma 와 연결구조를 가진 struct anon_vma_chain 을 가짐(blue : same_anon_vma / red : same_vma)



- process 가 fork 되어 새로운 vma 를 가지게 되면 parent 의 anon_vma 와 연결하기 위한 새로운 anon_vma_chain 이 생성되어 부모의 anon_vma 와 연결되고, 자신의 anon_vma 와 연결될 anon_vma_chain 이 새로 생성되어 자식 process 의 vm 과 또 연결됨



```
struct anon_vma_chain {
    struct vm_area_struct *vma;
    struct anon_vma *anon_vma;
    struct list_head same_vma; /* locked by mmap_sem & page_table_lock */
    struct list_head same_anon_vma; /* locked by anon_vma->mutex */
};
```



Reverse Mapping

- Reverse mapping history

- Step 6

- mprotect 시 vma 를 same_anon_vma 에서 제거하는 과정 등에서 same_anon_vma linked list 가 list 이기 때문에 원하는 vma 를 찾는데 오래 걸린다는 점을 해결하기 위해 수정

```
struct anon_vma_chain {  
    struct vm_area_struct *vma;  
    struct anon_vma *anon_vma;  
    struct list_head same_vma; /* locked by mmap_sem & page_table_lock */  
    struct list_head same_anon_vma; /* locked by anon_vma->mutex */  
};
```

v3.6.11



single list
→ interval tree

```
struct anon_vma_chain {  
    struct vm_area_struct *vma;  
    struct anon_vma *anon_vma;  
    struct list_head same_vma; /* locked by mmap_sem & page_table_lock */  
    struct rb_node rb; /* locked by anon_vma->mutex */  
    unsigned long rb_subtree_last;  
#ifdef CONFIG_DEBUG_VM_RB  
    unsigned long cached_vma_start, cached_vma_last;  
#endif  
};
```

v3.7

손주형

```
struct address_space {  
    struct inode *host; /* owner: inode, block_device */  
    struct radix_tree_root page_tree; /* radix tree of all pages */  
    spinlock_t tree_lock; /* and lock protecting it */  
    unsigned int i_mmap_writable; /* count VM_SHARED mappings */  
    struct prio_tree_root i_mmap; /* tree of private and shared mappings */  
    struct list_head i_mmap_nonlinear; /* list VM_NONLINEAR mappings */  
    struct mutex i_mmap_mutex; /* protect tree, count, list */  
/* Protected by tree_lock together with the radix tree */  
    unsigned long nrpages; /* number of total pages */  
    pgoff_t writeback_index; /* writeback starts here */  
    const struct address_space_operations *a_ops; /* methods */  
    unsigned long flags; /* error bits/gfp mask */  
    struct backing_dev_info *backing_dev_info; /* device readahead, etc */  
    spinlock_t private_lock; /* for use by the address_space */  
    struct list_head private_list; /* ditto */  
    struct address_space *assoc_mapping; /* ditto */  
} __attribute__((aligned(sizeof(long))));
```

v3.6.11



priority tree
→ interval tree

```
struct address_space {  
    struct inode *host; /* owner: inode, block_device */  
    struct radix_tree_root page_tree; /* radix tree of all pages */  
    spinlock_t tree_lock; /* and lock protecting it */  
    unsigned int i_mmap_writable; /* count VM_SHARED mappings */  
    struct rb_root i_mmap; /* tree of private and shared mappings */  
    struct list_head i_mmap_nonlinear; /* list VM_NONLINEAR mappings */  
    struct mutex i_mmap_mutex; /* protect tree, count, list */  
/* Protected by tree_lock together with the radix tree */  
    unsigned long nrpages; /* number of total pages */  
    pgoff_t writeback_index; /* writeback starts here */  
    const struct address_space_operations *a_ops; /* methods */  
    unsigned long flags; /* error bits/gfp mask */  
    struct backing_dev_info *backing_dev_info; /* device readahead, etc */  
    spinlock_t private_lock; /* for use by the address_space */  
    struct list_head private_list; /* ditto */  
    struct address_space *assoc_mapping; /* ditto */  
} __attribute__((aligned(sizeof(long))));
```

v3.7

Embedded System Lab. 

Reverse Mapping

- Reverse mapping data structure

- o anon_vma_chain

```
struct anon_vma_chain {
    struct vm_area_struct *vma;
    struct anon_vma *anon_vma;
    struct list_head same_vma; /* locked by mmap_sem & page_table_lock */
    struct rb_node rb;         /* locked by anon_vma->rwsem */
    unsigned long rb_subtree_last;
#endif CONFIG_DEBUG_VM_RB
    unsigned long cached_vma_start, cached_vma_last;
#endif
};
```

- 하나의 vma 를 관리하는 data structure 로 interval tree 의 각 node 에 해당
 - rb_subtree_max 를 통해 interval tree 관리
 - fork 되면 부모의 anon_vma 와 자식의 vma 연결용
anon_vma_chain 이 하나 더 생김

- o anon_vma

- 하나의 process 당 한 개의 anon_vma 가 존재 (A process 가 사용하는 page 들은 모두 A process 의 anon_vma 를 가리킴.)
 - fork 될 때마다 자식 process 의 anon_vma 가 부모 process 를 parent 로 설정하며 최고 부모 process 가 root 에 해당.(일반 tree 형식)
 - anon_vma_chain 의 interval tree 의 root node 가리킴
 - reference count : root node 를 가리키는 count
 - degree : 해당 node 를 가리키는 count

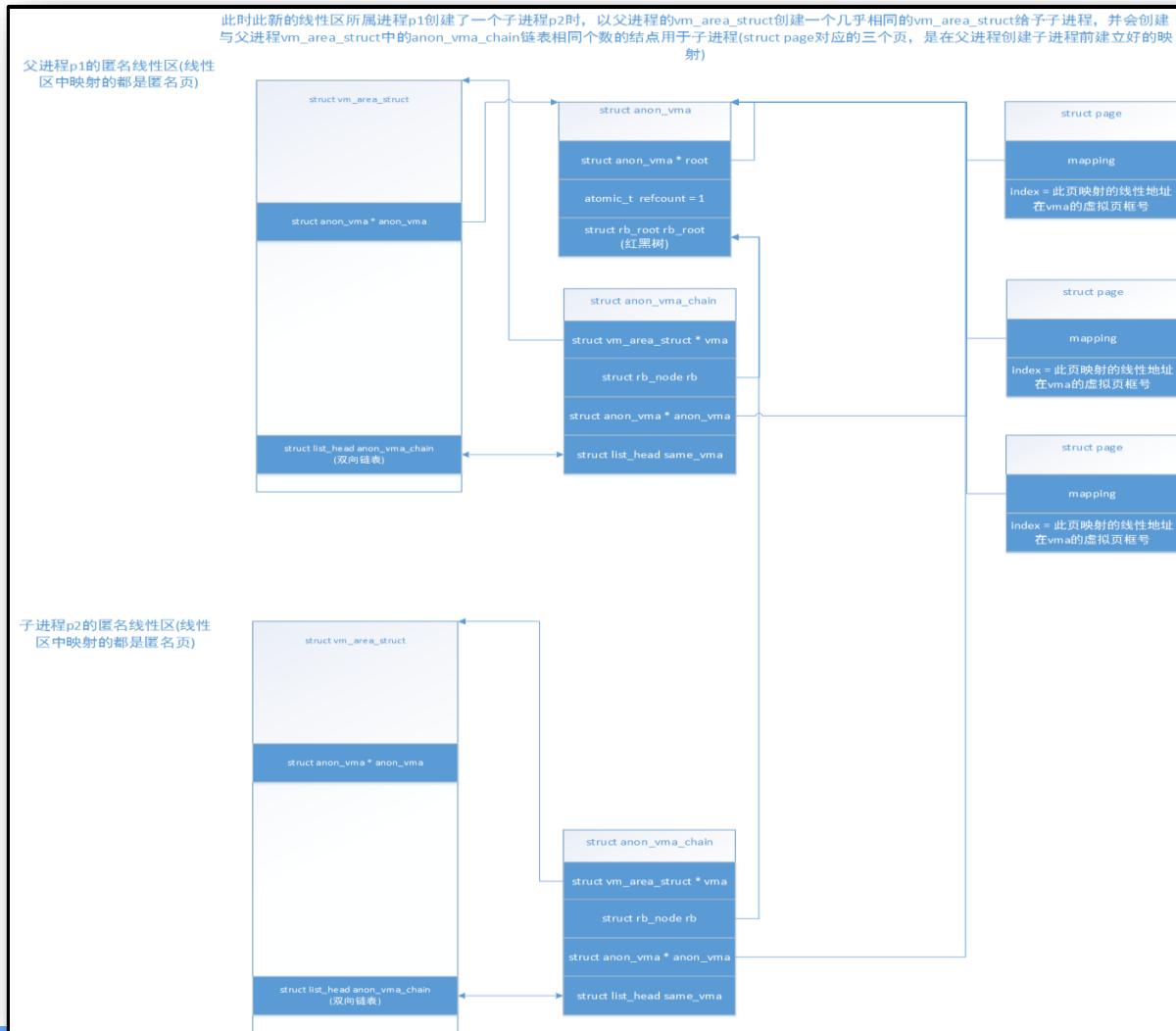
```

struct anon_vma {
    struct anon_vma *root;      /* Root of this anon_vma tree */
    // 현재 process 와 관련된 process 들의 anon_vma tree 에서
    // 최초 anon_vma 를 최초 process 의 anon_vma
    // red black ?
    struct rw_semaphore rwsem; /* W: modification, R: walking the list */
/*
 * The refcount is taken on an anon_vma when there is no
 * guarantee that the vma of page tables will exist for
 * the duration of the operation. A caller that takes
 * the reference is responsible for clearing up the
 * anon_vma if they are the last user on release
 */
atomic_t refcount;
// 현재 anon_vma 를 가리키는 child anon_vma 의 개수 .
/*
 * Count of child anon_vmas and VMAs which points to this anon_vma.
 * This counter is used for making decision about reusing anon_vma
 * instead of forking new one. See comments in function anon_vma_clone.
 */
unsigned degree;
// anon_vma 와 연결된 child anon_vma
//
// P->C0->C1->C2, C1->C3, C1->C4 으로 프로세스가 생겼다 하면
//
// parent process          P_anon_vma
//                           |
//   -----
// child processes          |           |           |
//   C0_anon_vma            C1_anon_vma        C2_anon_vma
//                           |
//   -----
//                           |           |           |
//   C3_anon_vma            C4_anon_vma
//
// 이 경우 상황이면 C0~C4 의 root 는 P_anon_vma
// C3_anon_vma 의 parent 는 C1_anon_vma
// P_anon_vma 의 refcount 는 5
// C1_anon_vma 의 degree 는 2
// P_anon_vma 의 degree 는 3
//
struct anon_vma *parent; /* Parent of this anon_vma */
// anon_vma tree 에서 현재 anon_vma 의 parent anon_vma 즉
// 현재 process 를 생성한 parent process 의 anon_vma
/*
 * NOTE: the LSB of the rb_root.rb_node is set by
 * mm_take_all_locks() _after_ taking the above lock. So the
 * rb_root must only be read/written after taking the above lock
 * to be sure to see a valid next pointer. The LSB bit itself
 * is serialized by a system wide lock only visible to
 * mm_take_all_locks() (mm_all_locks_mutex).
 */
struct rb_root rb_root; /* Interval tree of private "related" vmas */
// reverse mapping 을 위한 interval tree 로 여기에
// anon_vma_chain 를 이 Interval tree 구조로 연결되어 있음
};

};
```

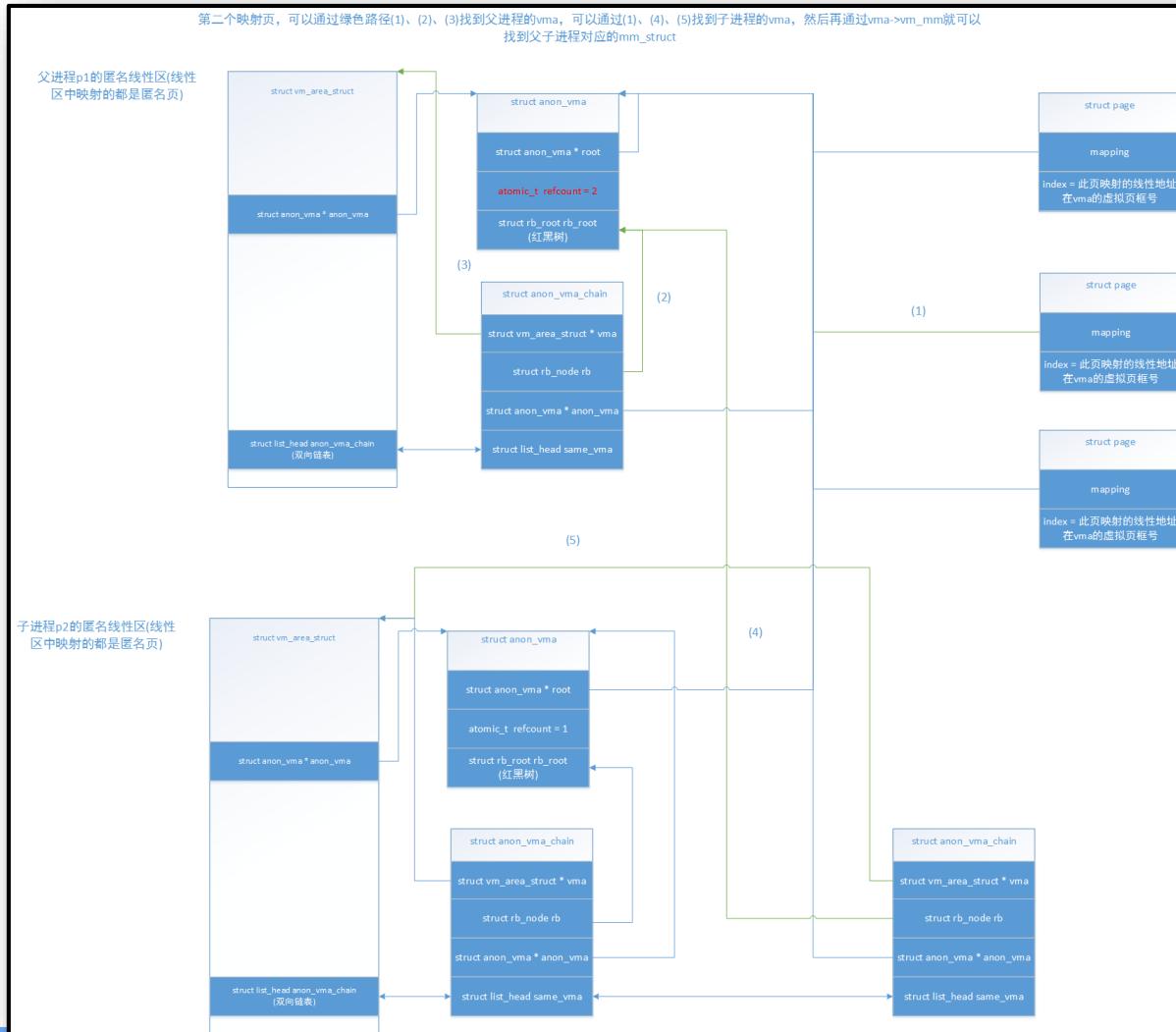
Reverse Mapping

- Reverse mapping diagram
 - fork 시 anon_vma 생성/연결 전.(anon_vma_fork)



Reverse Mapping

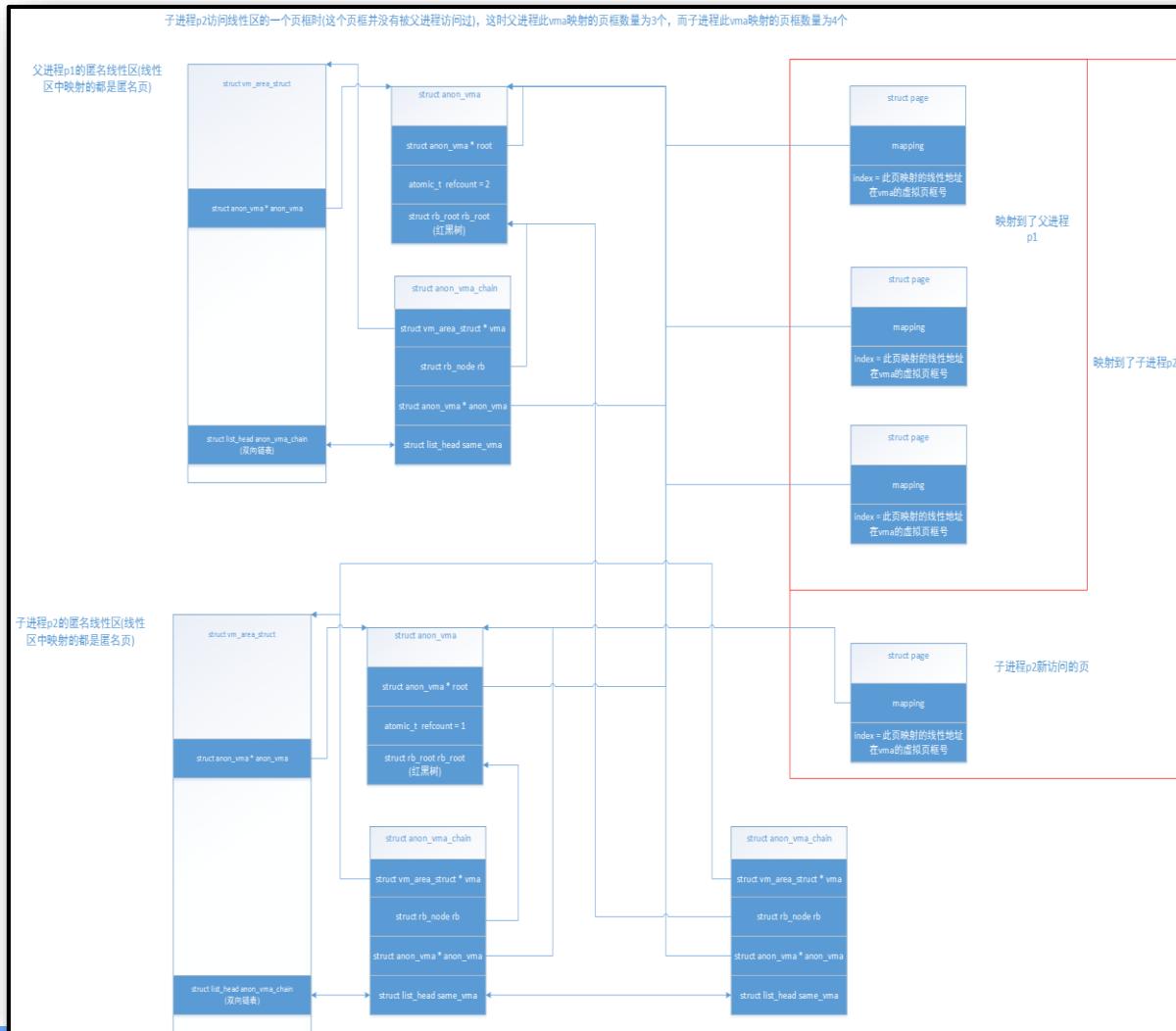
- Reverse mapping diagram
 - fork 시 anon_vma 생성/연결 후.(anon_vma_fork), COW 에 의해 같은 page 공유 상태



Reverse Mapping

- Reverse mapping diagram

- fork 시 anon_vma 생성/연결 후.(anon_vma_fork), page 에 대한 write 수행 후 자식 process 를 위한 새로운 page 생성 된 상태



Appendix – Interval Tree

● Interval Tree

- 기존 rbtree 에서는 직접적으로 range 범위 관리가 불가능 하여 range 를 기반으로 rbtree 구성 시, 현재 tree 에 range 에 해당하는 것이 존재하는지 일일이 해당 기능을 구현해 주어야 함.
- augmented rbtree 를 기반으로 rbtree 내에 특정 값이 추가로 관리되도록 하여 range 를 쉽게 관리 가능
- 기본 rbtree 와 달리 insert, remove iterate 를 제공.
- Kernel 내의 아래와 같은 곳에서 사용
 - file backed page reverse mapping 에서의 anon_vma_struct 관리
 - anonymous page reverse mapping 에서의 vm_area_struct 관리
 - gpu 에서 ...

● Algorithm

- insert : start 를 기준으로 BST 와 같은 방법으로 수행
 - vma 관리의 경우, 현재 virtual address 의 page 단위 offset
- search : 현재 node 의 range 와 겹치지 않는다면 __subtree_last 를 기준으로
 - vma 관리의 경우, 현재 virtual address 의 page 단위로 그전 vma 의 끝 offset 과의 차이

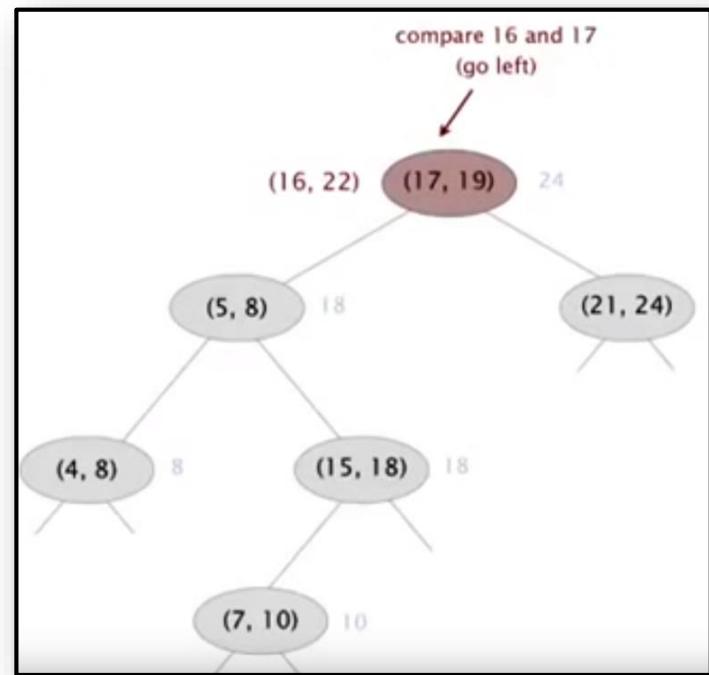
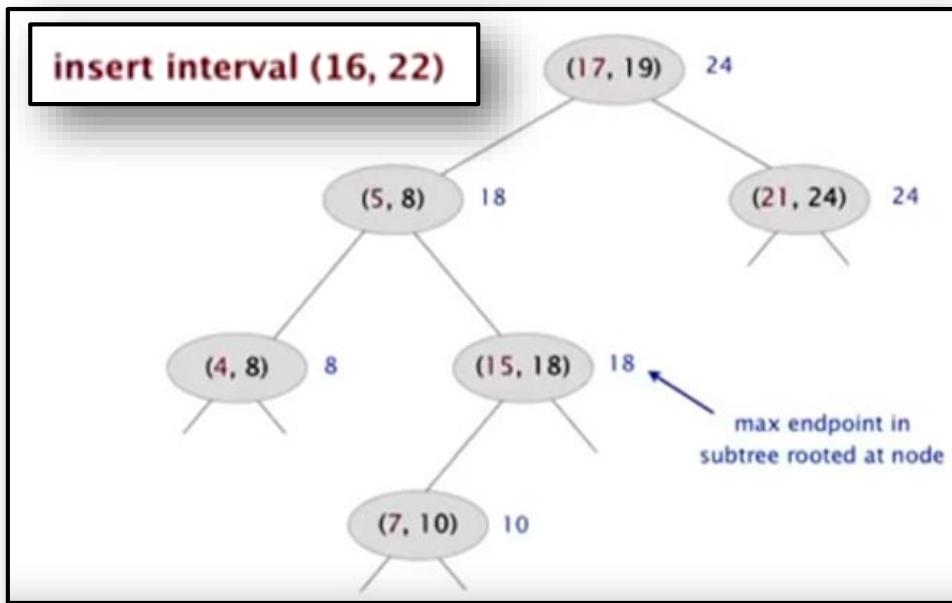
```
struct interval_tree_node {  
    struct rb_node rb;  
    // red black tree node  
    unsigned long start; /* Start of interval */  
    // 현재 node range 의 start  
    unsigned long last; /* Last location _in_ interval */  
    // 현재 node range 의 end  
    unsigned long __subtree_last;  
    // red black tree 에서 child subtree 의 가장 큰 last 값  
};
```

```
struct anon_vma_chain {  
    struct vm_area_struct *vma;  
    struct anon_vma *anon_vma;  
    struct list_head same_vma; /* locked by mmap_sem & page_table_lock */  
    struct rb_node rb; /* locked by anon_vma->rwssem */  
    unsigned long rb_subtree_last;  
#ifdef CONFIG_DEBUG_VM_RB  
    unsigned long cached_vma_start, cached_vma_last;  
#endif  
};
```

Appendix – Interval Tree

- Algorithm

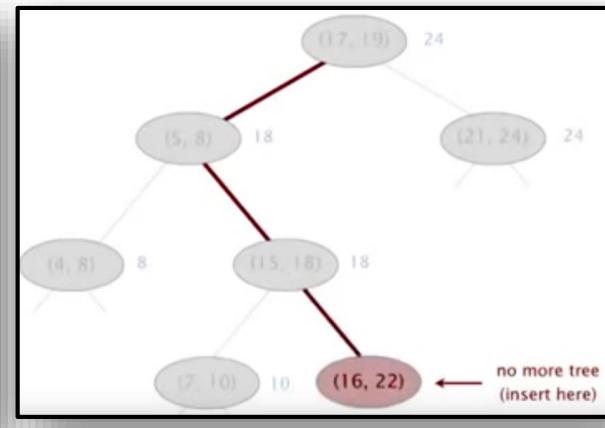
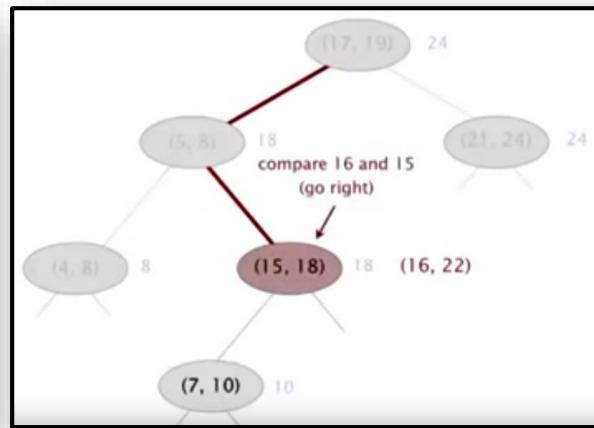
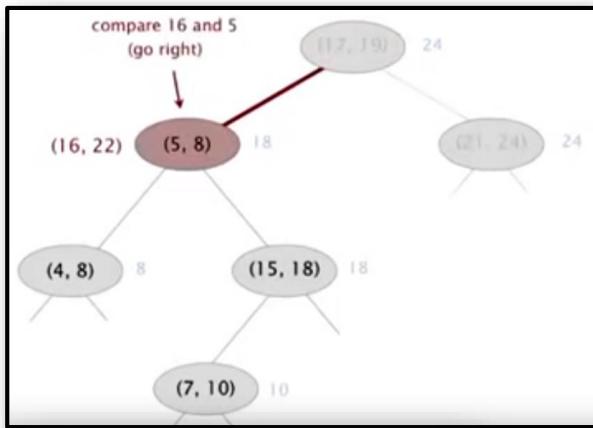
- insert : range의 start 를 key 값으로 하여 기준으로 BST 와 같은 방법으로 수행 및 augment 값 update



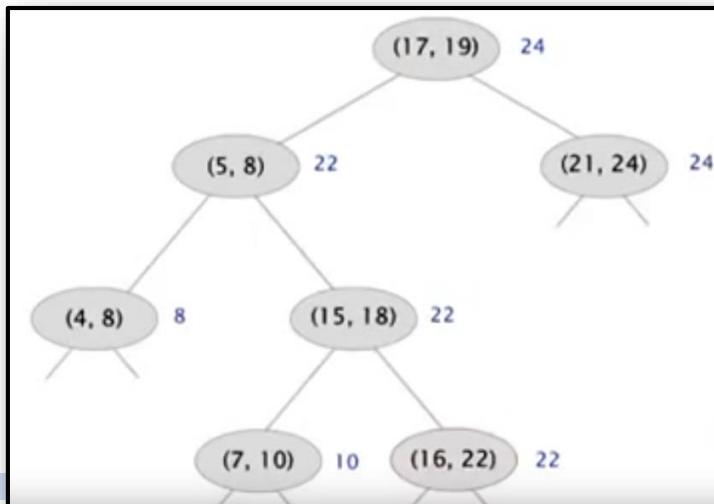
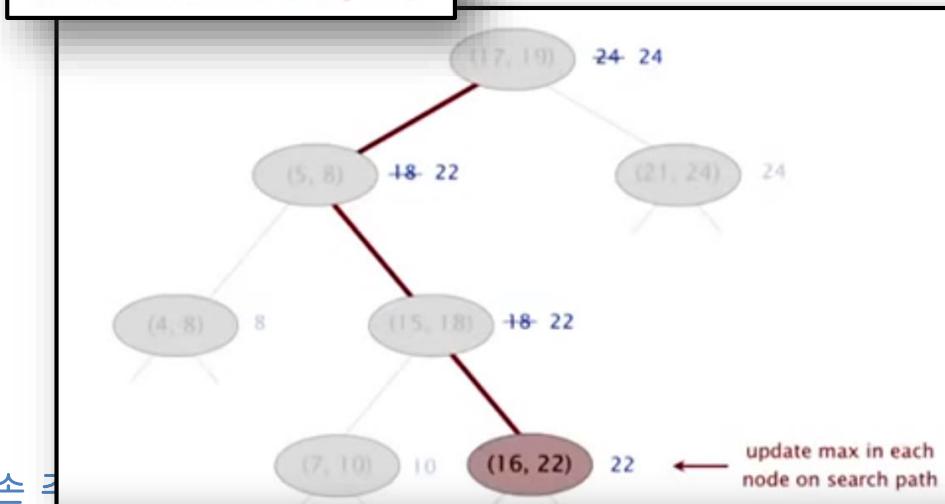
Appendix – Interval Tree

- Algorithm

- insert : range의 start 를 key 값으로 하여 기준으로 BST 와 같은 방법으로 수행 및 augment 값 update



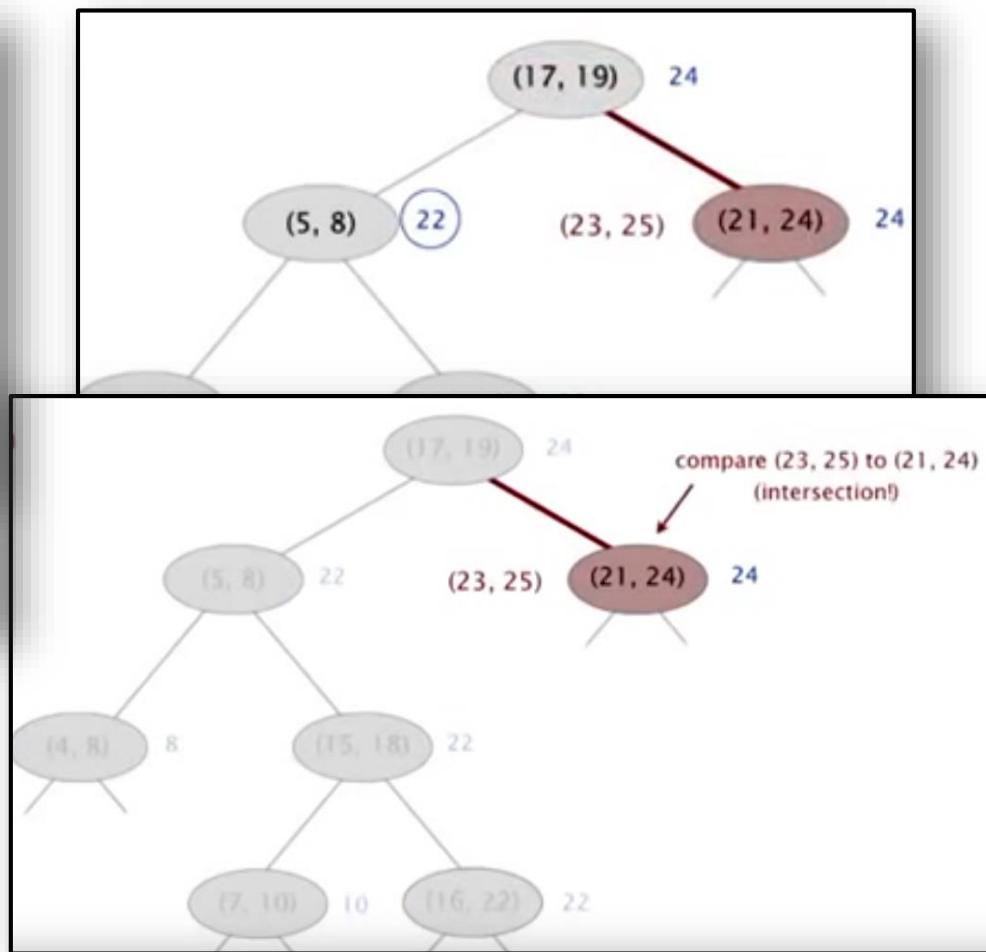
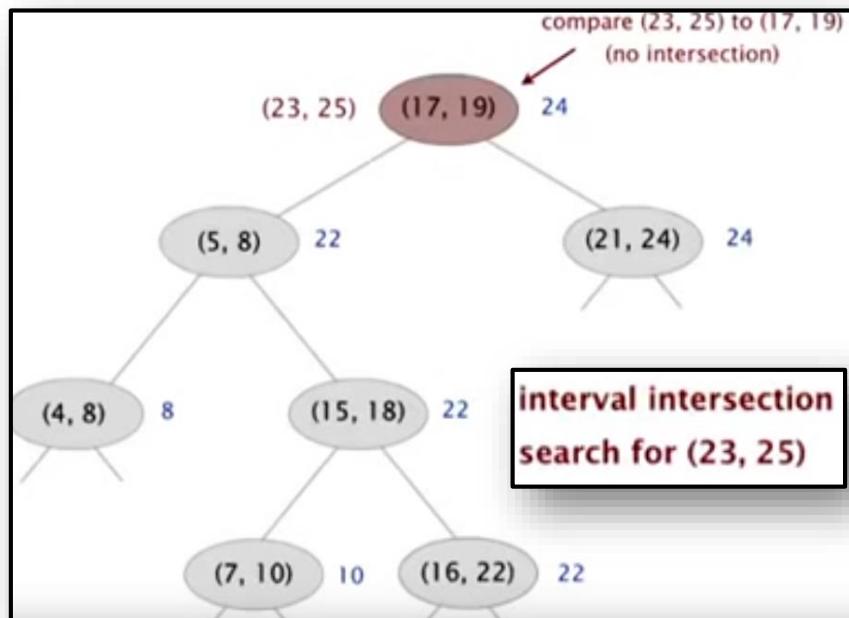
insert interval $(16, 22)$



Appendix – Interval Tree

- Algorithm

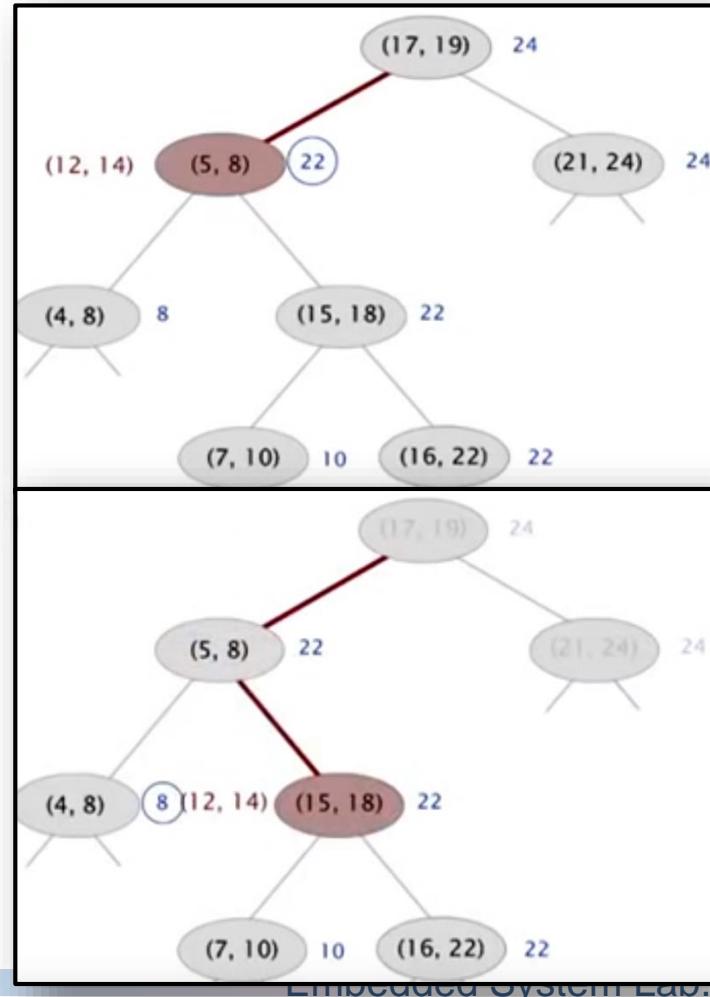
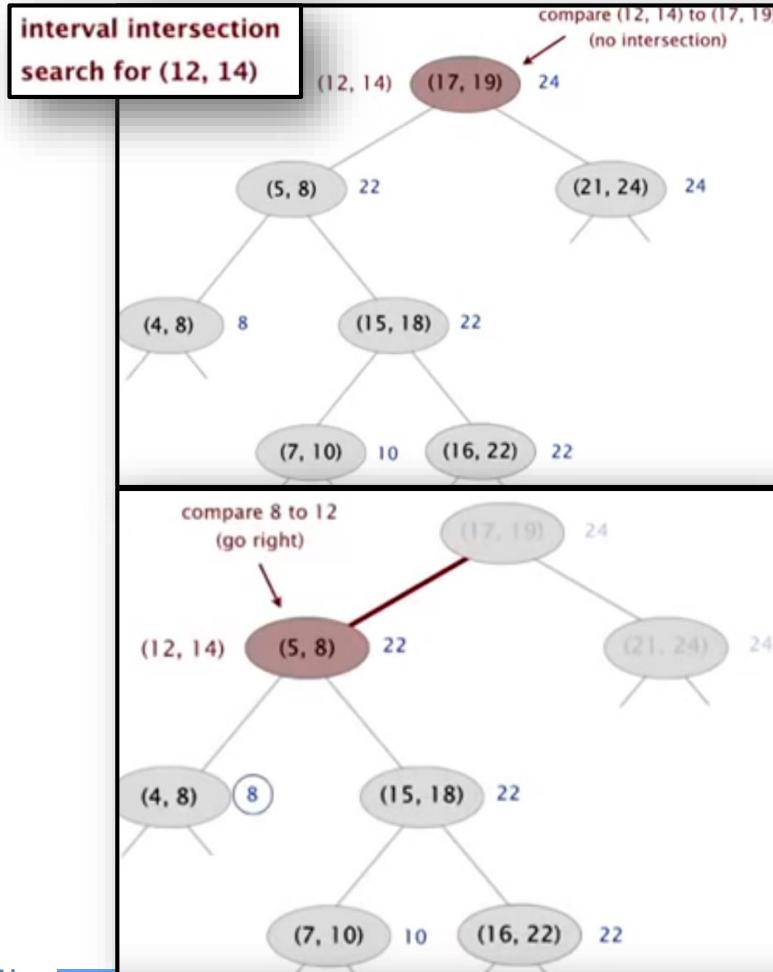
- search : 현재 node range 에 해당되지 않는다면 left child node 의 sub tree max 값과 insert 하려는 range 의 start 를 비교하여 start 가 더 작다면 left child 로 이동, 크다면 right child 로 이동 및 augment 값 update



Appendix – Interval Tree

Algorithm

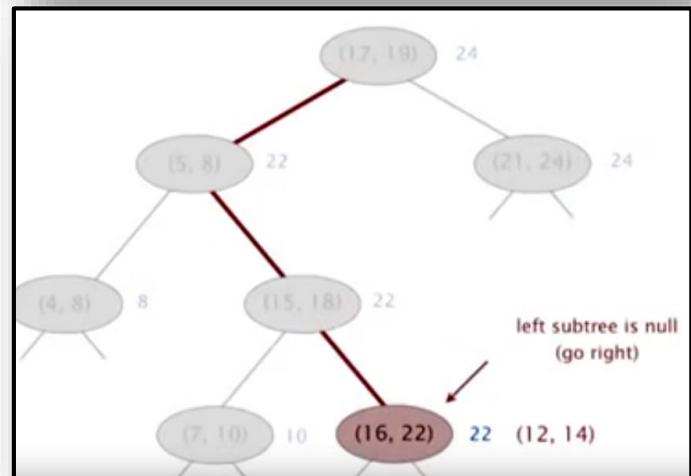
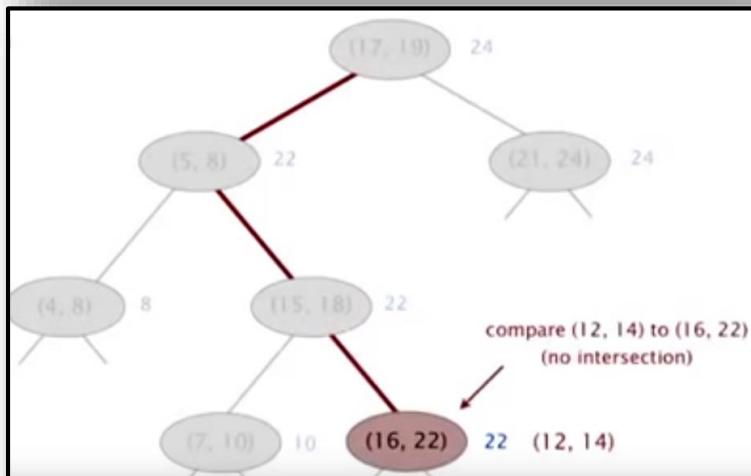
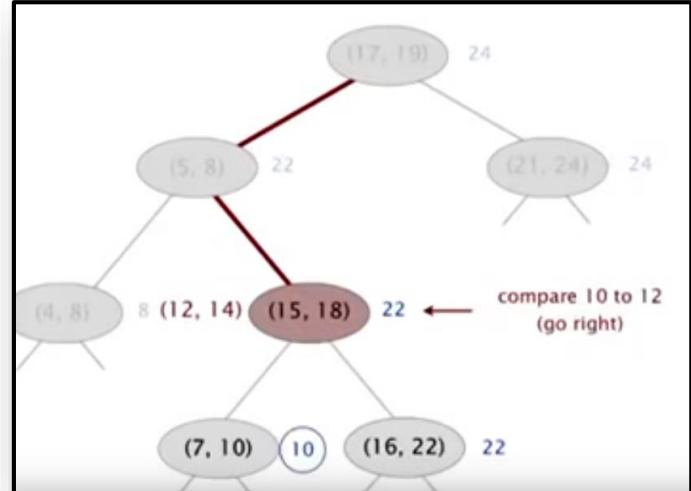
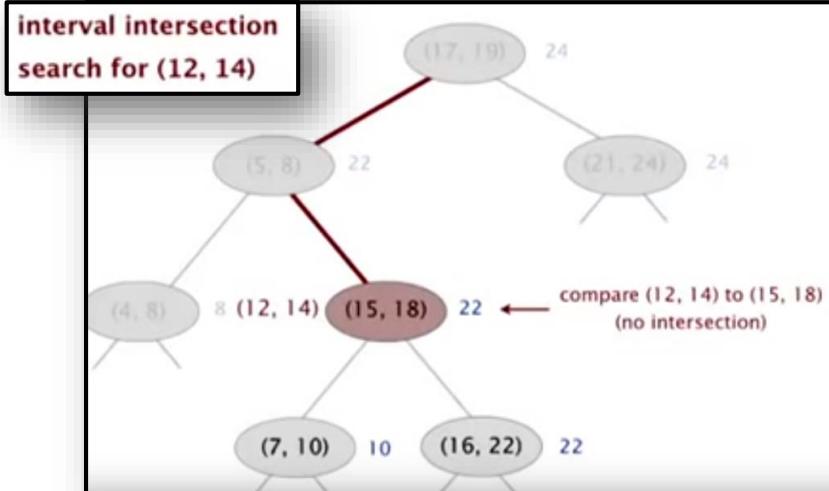
- search : 현재 node range 에 해당되지 않는다면 left child node 의 sub tree max 값과 insert 하려는 range 의 start 를 비교하여 start 가 더 작다면 left child 로 이동, 크다면 right child 로 이동 및 augment 값 update



Appendix – Interval Tree

- Algorithm

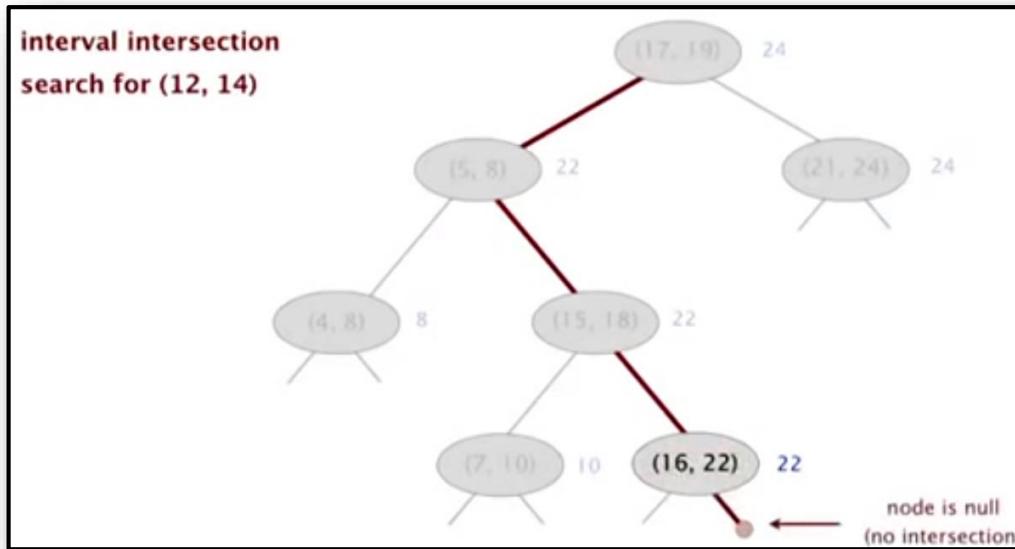
- search : 현재 node range 에 해당되지 않는다면 left child node 의 sub tree max 값과 insert 하려는 range 의 start 를 비교하여 start 를 비교하여 start 가 더 작다면 left child 로 이동, 크다면 right child 로 이동 및 augment 값 update



Appendix – Interval Tree

- Algorithm

- search : 현재 node range 에 해당되지 않는다면 left child node 의 sub tree max 값과 insert 하려는 range 의 start 를 비교하여 start 가 더 작다면 left child 로 이동, 크다면 right child 로 이동 및 augment 값 update



Appendix – Interval Tree

Interval Tree kernel API

- INTERVAL_TREE_DEFINE 함수 선언 & 정의
- ISTART, ILAST 매크로
- RB_DECLARE_CALLBACKS 로 interval tree에서 사용될 augment tree 생성
- IPREFIX에 전달된 문자열을 prefix로 가지는 아래 함수들이 생성됨
 - compute_subtree_last
 - _insert
 - _remove
 - _subtree_search
 - _iter_first
 - _iter_next

```
#define INTERVAL_TREE_DEFINE(ITSTRUCT, ITRB, ITTYPE, ITSUBTREE, \
                           ITSTART, ILAST, ITSTATIC, ITPREFIX) \
                           \ \
/* Callbacks for augmented rbtree insert and remove */ \
                           \ \
static inline ITTYPE ITPREFIX ## _compute_subtree_last(ITSTRUCT *node) \
{ \
    ITTYPE max = ILAST(node), subtree_last; \
    if (node->ITRB.rb_left) { \
        subtree_last = rb_entry(node->ITRB.rb_left, \
                               ITSTRUCT, ITRB)->ITSUBTREE; \
        if (max < subtree_last) \
            max = subtree_last; \
    } \
    if (node->ITRB.rb_right) { \
        subtree_last = rb_entry(node->ITRB.rb_right, \
                               ITSTRUCT, ITRB)->ITSUBTREE; \
        if (max < subtree_last) \
            max = subtree_last; \
    } \
    return max; \
} \
\ \
RB_DECLARE_CALLBACKS(static, ITPREFIX ## _augment, ITSTRUCT, ITRB, \
                     ITTYPE, ITSUBTREE, ITPREFIX ## _compute_subtree_last) \
\ \
/* Insert / remove interval nodes from the tree */ \
\ \
ITSTATIC void ITPREFIX ## _insert(ITSTRUCT *node, struct rb_root *root) \
{ \
    struct rb_node **link = &root->rb_node, *rb_parent = NULL; \
    ITTYPE start = ITSTART(node), last = ILAST(node); \
    ITSTRUCT *parent; \
    \ \
    while (*link) { \
        rb_parent = *link; \
        parent = rb_entry(rb_parent, ITSTRUCT, ITRB); \
        if (parent->ITSUBTREE < last) \
            parent->ITSUBTREE = last; \
        if (start < ITSTART(parent)) \
            link = &parent->ITRB.rb_left; \
        else \
            link = &parent->ITRB.rb_right; \
    } \
    node->ITSUBTREE = last; \
    rb_link_node(&node->ITRB, rb_parent, link); \
    rb_insert_augmented(&node->ITRB, root, &ITPREFIX ## _augment); \
} \
\ \
ITSTATIC void ITPREFIX ## _remove(ITSTRUCT *node, struct rb_root *root) \
{ \
    rb_erase_augmented(&node->ITRB, root, &ITPREFIX ## _augment); \
}
```

Appendix – Interval Tree

Interval Tree kernel API

- INTERVAL_TREE_DEFINE 함수 선언 & 정의
- ISTART, ILAST 매크로
 - vma 를 통해 interval tree 에 사용될 start, end 기준 값 계산(page 단위 offset)
- RB_DECLARE_CALLBACK 로 interval tree 에서 사용될 augment tree 생성
- IPREFIX에 전달된 문자열을 prefix 로 가지는 아래 함수들이 생성됨
 - compute_subtree_last
 - _insert
 - _remove
 - _subtree_search
 - _iter_first
 - _iter_next

```
INTERVAL_TREE_DEFINE(struct vm_area_struct, shared.rb,
                     unsigned long, shared.rb_subtree_last,
                     vma_start_pgoff, vma_last_pgoff, vma_interval_tree)

static inline unsigned long vma_start_pgoff(struct vm_area_struct *v)
{
    return v->vm_pgoff;
}

static inline unsigned long vma_last_pgoff(struct vm_area_struct *v)
{
    return v->vm_pgoff + ((v->vm_end - v->vm_start) >> PAGE_SHIFT) - 1;
}

if (max < subtree_last)
    max = subtree_last;
return max;

INTERVAL_TREE_DEFINE(struct anon_vma_chain, rb, unsigned long, rb_subtree_last,
                     avc_start_pgoff, avc_last_pgoff,
                     static inline, __anon_vma_interval_tree)

static inline unsigned long avc_start_pgoff(struct anon_vma_chain *avc)
{
    // rmap 의 interval tree 를 관리 하는 기준인 start ~ end 에서 start 를
    // 가져오는 함수 .
    // start 는 vm_start 가 아니라 vm_pgoff 이다 .
    // page 단위로 몇 번째 page 위치에 존재하는지를 의미 .
    // insert 할 때 start 를 기준으로 BST 처럼 동작
    return vma_start_pgoff(avc->vma);
}

static inline unsigned long avc_last_pgoff(struct anon_vma_chain *avc)
{
    // rmap 의 interval tree 를 관리 하는 기준인 start ~ last 에서 last 를
    // 가져오는 함수 .
    // last 는 vm_start ~ vm_end 까지의 page 단위 크기에 현재
    // vma 의 page 단위 offset 시작 주소를 더한 값이다 .
    // subtree_last 를 통해 interval tree search 시에 사용 .
    return vma_last_pgoff(avc->vma);
}

ITSTATIC void ITPREFIX ## _remove(ITSTRUCT *node, struct rb_root *root)
{
    rb_erase_augmented(&node->ITRB, root, &ITPREFIX ## _augment);
}
```

Appendix – Interval

- Interval Tree kernel API

- INTERVAL_TREE_DEFINE 함수 선언 & 정의
 - ISTART, ILAST 매크로
 - vma 를 통해 interval tree 에 사용될 start, end 기준 값 계산(page 단위 offset)
 - RB_DECLARE_CALLBACK 로 interval tree 에서 사용될 augment tree 생성
 - IPREFIX에 전달된 문자열을 prefix 로 가지는 아래 함수들이 생성됨
 - compute_subtree_last
 - left child, right child 의 subtree_last 값 중 큰 값을 현재 값으로 설정
 - augment 함수로 등록
 - _insert
 - _remove
 - _subtree_search
 - _iter_first
 - _iter_next

Appendix – Interval Tree

Interval Tree kernel API

- INTERVAL_TREE_DEFINE 함수 선언 & 정의
- ISTART, ILAST 매크로
 - vma 를 통해 interval tree 에 사용될 start, end 기준 값 계산(page 단위 offset)
- RB_DECLARE_CALLBACK 로 interval tree 에서 사용될 augment tree 생성
- IPREFIX에 전달된 문자열을 prefix 로 가지는 아래 함수들이 생성됨
 - compute_subtree_last
 - left child, right child 의 subtree_last 값 중 큰 값을 현재 값으로 설정
 - augment 함수로 등록
 - _insert
 - insert 하며 순회하는 node 의 subtree 값보다 last 가 클 경우, last 를 update
 - start 를 기준으로 child 로 이동
 - _remove
 - remove & balance
 - _subtree_search
 - _iter_first
 - _iter_next

```
#define INTERVAL_TREE_DEFINE(ITSTRUCT, ITRB, ITTYPE, ITSUBTREE, \
                           ITSTART, ITLAST, ITSTATIC, ITPREFIX) \
                           \
static inline void __anon_vma_interval_tree_insert(struct anon_vma_chain *node, struct rb_root *root) \
{ \
    struct rb_node **link = &root->rb_node, *rb_parent = NULL; \
    unsigned long start = avc_start_pgoff(node), last = avc_last_pgoff(node); \
    struct anon_vma_chain *parent; \
    \
    while (*link) { \
        rb_parent = *link; \
        parent = rb_entry(rb_parent, struct anon_vma_chain, rb); \
        if (parent->rb_subtree_last < last) \
            parent->rb_subtree_last = last; \
        /* rmap - 현재 범위의 last 값이 rb_subtree_last 값보다 크다면 \
         * 즉 현재 rbtree 의 max 값보다 지금 insert 하려는 값이 크다면 \
         * augment 값 update하면서 child 로 내려감 */ \
        if (start < avc_start_pgoff(parent)) \
            link = &parent->rb.rb_left; \
        else \
            link = &parent->rb.rb_right; \
        /* rmap - start 를 기준으로 BST 형식으로 insert 수행 */ \
    } \
    \
    node->rb_subtree_last = last; \
    rb_link_node(&node->rb, rb_parent, link); \
    rb_insert_augmented(&node->rb, root, &__anon_vma_interval_tree ## _augment); \
} \
 \
RB_DECLARE_CALLBACKS(static, ITPREFIX ## _augment, ITSTRUCT, ITRB, \
                     ITTYPE, ITSUBTREE, ITPREFIX ## _compute_subtree_last) \
                     \
/* Insert / remove interval nodes from the tree */ \
 \
ITSTATIC void ITPREFIX ## _insert(ITSTRUCT *node, struct rb_root *root) \
{ \
    struct rb_node **link = &root->rb_node, *rb_parent = NULL; \
    ITTYPE start = ITSTART(node), last = ITLAST(node); \
    ITSTRUCT *parent; \
    \
    while (*link) { \
        rb_parent = *link; \
        parent = rb_entry(rb_parent, ITSTRUCT, ITRB); \
        if (parent->ITSUBTREE < last) \
            link = &parent->rb.rb_right; \
        else \
            link = &parent->rb.rb_left; \
    } \
    \
    node->ITSUBTREE = last; \
    rb_link_node(&node->ITRB, rb_parent, link); \
    rb_insert_augmented(&node->ITRB, root, &ITPREFIX ## _augment); \
} \
 \
ITSTATIC void ITPREFIX ## _remove(ITSTRUCT *node, struct rb_root *root) \
{ \
    rb_erase_augmented(&node->ITRB, root, &ITPREFIX ## _augment); \
}
```

Appendix – Interval

- Interval Tree kernel API

- INTERVAL_TREE_DEFINE 함수 선언 & 정의
 - ISTART, ILAST 매크로
 - vma 를 통해 interval tree 에 사용될 start, end 기준 값 계산(page 단위 offset)
 - RB_DECLARE_CALLBACK 로 interval tree 에서 사용될 augment tree 생성
 - IPREFIX에 전달된 문자열을 prefix 로 가지는 아래 함수들이 생성됨
 - compute_subtree_last
 - left child, right child 의 subtree_last 값 중 큰 값을 현재 값으로 설정
 - augment 함수로 등록
 - _insert
 - insert 하며 순회하는 node 의 subtree 값보다 last 가 클 경우, last 를 update
 - start 를 기준으로 child 로 이동
 - _remove
 - remove & balance
 - _subtree_search
 - _iter_first
 - _iter_next

```
define INTERVAL_TREE_DEFINE(ITSTRUCT, ITRB, ITTYPE, ITSUBTREE,  
    ITSTART, ITLAST, ITSTATIC, ITPREFIX) \\\n  
  
static inline void __anon_vma_interval_tree_insert(struct anon_vma_chain *node, struct rb_root *root)  
{  
    struct rb_node **link = &root->rb_node, *rb_parent = NULL;  
    unsigned long start = avc_start_pgoff(node), last = avc_last_pgoff(node);  
    struct anon_vma_chain *parent;  
  
    while (*link) {  
        rb_parent = *link;  
        parent = rb_entry(rb_parent, struct anon_vma_chain, rb);  
        if (parent->rb_subtree_last < last)  
            parent->rb_subtree_last = last;  
        /* rmap - 현재 범위의 last 값이 rb_subtree_last 값보다 크다면  
         즉 현재 rbtree 의 max 값보다 지금 insert 하려는 값이 크다면  
         augment 값 update하면서 child로 내려감 */  
        if (start < avc_start_pgoff(parent))  
            link = &parent->rb.rb_left;  
        else  
            link = &parent->rb.rb_right;  
        /* rmap - start 를 기준으로 BST 형식으로 insert 수행 */  
    }  
  
    node->rb_subtree_last = last;  
    rb_link_node(&node->rb, rb_parent, link);  
    rb_insert_augmented(&node->rb, root, &__anon_vma_interval_tree ## _augment);  
}  
  
void anon_vma_interval_tree_insert(struct anon_vma_chain *node,  
                                    struct rb_root *root)  
{  
#ifdef CONFIG_DEBUG_VM_RB  
    node->cached_vma_start = avc_start_pgoff(node);  
    node->cached_vma_last = avc_last_pgoff(node);  
#endif  
    __anon_vma_interval_tree_insert(node, root);  
}  
    rb_parent = *link;  
    parent = rb_entry(rb_parent, ITSTRUCT, ITRB);  
    if (parent->ITSUBTREE < last)  

```

Appendix – Interval Tree

Interval Tree kernel API

- IPREFIX에 전달된 문자열을 prefix로 가지는 아래 함수들이 생성됨
 - _subtree_search
 - _iter_first

```
static ITSTRUCT *
ITPREFIX ## _subtree_search(ITSTRUCT *node, ITTYPE start, ITTYPE last)
{
    while (true) {
        /*
         * Loop invariant: start <= node->ITSUBTREE
         * (Cond2 is satisfied by one of the subtree nodes)
         */
        if (node->ITRB.rb_left) {
            ITSTRUCT *left = rb_entry(node->ITRB.rb_left,
                                       ITSTRUCT, ITRB);
            if (start <= left->ITSUBTREE) {
                /*
                 * Some nodes in left subtree satisfy Cond2.
                 * Iterate to find the leftmost such node N.
                 * If it also satisfies Cond1, that's the
                 * match we are looking for. Otherwise, there
                 * is no matching interval as nodes to the
                 * right of N can't satisfy Cond1 either.
                 */
                node = left;
                continue;
            }
        }
        if (ITSTART(node) <= last) { /* Cond1 */
            if (start <= ITLAST(node)) /* Cond2 */
                return node; /* node is leftmost match */
            if (node->ITRB.rb_right) {
                node = rb_entry(node->ITRB.rb_right,
                               ITSTRUCT, ITRB);
                if (start <= node->ITSUBTREE)
                    continue;
            }
        }
        return NULL; /* No match */
    }
}

ITSTATIC ITSTRUCT *
ITPREFIX ## _iter_first(struct rb_root *root, ITTYPE start, ITTYPE last)
{
    ITSTRUCT *node;
    if (!root->rb_node)
        return NULL;
    node = rb_entry(root->rb_node, ITSTRUCT, ITRB);
    if (node->ITSUBTREE < start)
        return NULL;
    return ITPREFIX ## _subtree_search(node, start, last);
}
```

Appendix – Interval Tree

Interval Tree kernel API

- IPREFIX에 전달된 문자열을 prefix로 가지는 아래 함수들이 생성됨
 - _subtree_search
 - search 하려는 start와 left child의 subtree_last 비교하여 left 이동 여부 결정
 - 현재 node 범위 검사
 - right child 이동 여부 검사
 - _iter_first

```
static ITSTRUCT *
ITPREFIX ## _subtree_search(ITSTRUCT *node, ITTYPE start, ITTYPE last)
{
    while (true) {
        /*
         */
        static struct anon_vma_chain *
_anon_vma_interval_tree_subtree_search(struct anon_vma_chain *node, unsigned long start, unsigned long last)
{
    while (true) {
        /*
         * Loop invariant: start <= node->rb_subtree_last
         * (Cond2 is satisfied by one of the subtree nodes)
         */
        if (node->rb.rb_left) {
            struct anon_vma_chain *left = rb_entry(node->rb.rb_left,
                                         struct anon_vma_chain, rb);
            if (start <= left->rb_subtree_last) {
                /*
                 * Some nodes in left subtree satisfy Cond2.
                 * Iterate to find the leftmost such node N.
                 * If it also satisfies Cond1, that's the
                 * match we are looking for. Otherwise, there
                 * is no matching interval as nodes to the
                 * right of N can't satisfy Cond1 either.
                 */
                node = left;
                continue;
            }
            /* rmap - left child로 이동할 수 있는 경우, left로 이동
             * left child의 subtree_last 값 즉 left child subtree에서의
             * 가장 큰 last 값이 현재 start 보다 커서 left child subtree
             * 내에 걸리는 range가 있어야 함 */
        }
        if (avc_start_pgoff(node) <= last) { /* Cond1 */
            if (start <= avc_last_pgoff(node)) /* Cond2 */
                return node; /* node is leftmost match */
        }
        /* rmap - left child로 이동할 수 없다면 즉
         * left subtree의 범위 최대 값이 현재 범위 시작 값
         * 보다 작으므로 left subtree 내에 원하는 range가 없는
         * 상태임.
         * 이제 현재 node의 범위가 start ~ end 내에
         * 걸리는지 검사 */
        if (node->rb.rb_right) {
            node = rb_entry(node->rb.rb_right,
                            struct anon_vma_chain, rb);
            if (start <= node->rb_subtree_last)
                continue;
            /* rmap - right subtree의 범위 최대 값이 start 보다 작다면
             * 즉 right subtree에 range에 포함되는 값이 있다면 이동 */
        }
    }
    return NULL; /* No match */
}
node = rb_entry(root->rb_node, ITSTRUCT, ITRB);
if (node->ITSUBTREE < start)
    return NULL;
return ITPREFIX ## _subtree_search(node, start, last);
}
```

Appendix – Interval Tree

Interval Tree kernel API

- IPREFIX에 전달된 문자열을 prefix로 가지는 아래 함수들이 생성됨
 - _subtree_search
 - search 하려는 start와 left child의 subtree_last 비교하여 left 이동 여부 결정
 - 현재 node 범위 검사
 - right child 이동 여부 검사
 - _iter_first
 - 해당 범위 root에서 subtree_last 통해 검사 후 없다면 search

```
static ITSTRUCT *
ITPREFIX ## _subtree_search(ITSTRUCT *node, ITTYPE start, ITTYPE last)
{
    while (true) {
        /*
         * Loop invariant: start <= node->ITSUBTREE
         * (Cond2 is satisfied by one of the subtree nodes)
         */
        if (node->ITRB.rb_left) {
            ITSTRUCT *left = rb_entry(node->ITRB.rb_left,
                                         ITSTRUCT, ITRB);
            if (start <= left->ITSUBTREE) {
                /*
                 * Some nodes in left subtree satisfy Cond2.
                 * Iterate to find the leftmost such node N.
                 * If it also satisfies Cond1, that's the
                 * match we are looking for. Otherwise, there
                 * is no matching interval as nodes to the
                 * right of N can't satisfy Cond1 either.
                 */
                node = left;
                continue;
            }
        }
        if (ITSTART(node) <= last) { /* Cond1 */
```

```
static inline struct anon_vma_chain *
__anon_vma_interval_tree_iter_first(struct rb_root *root, unsigned long start, unsigned long last)
{
    struct anon_vma_chain *node;

    if (!root->rb_node)
        return NULL;
    node = rb_entry(root->rb_node, struct anon_vma_chain, rb);
    if (node->rb_subtree_last < start)
        return NULL;
    /* ramp - 무조건 search 하기 전에 먼저 현재 해당 범위에 해당되는 값이
     * 있는지 rb_subtree_last 비교해서 확인하고 search를 어김 */
    return __anon_vma_interval_tree_subtree_search(node, start, last);
}
```

```
struct anon_vma_chain *
anon_vma_interval_tree_iter_first(struct rb_root *root,
                                  unsigned long first, unsigned long last)
{
    return __anon_vma_interval_tree_iter_first(root, first, last);
}
```

Appendix – Interval Tree

● Interval Tree kernel API

- IPREFIX에 전달된 문자열을 prefix로 가지는 아래 함수들이 생성됨
 - _iter_next
 - 현재 node의 right child에서 원하는 range 값 검색
 - 없을 경우 parent의 left child에 붙어있는 node를 발견할 때 까지 상위로 이동
 - 찾은 parent의 range 검사
 - 없을 경우 위 과정 반복

```
ITSTATIC ITSTRUCT *
ITPREFIX ## _iter_next(ITSTRUCT *node, ITTYPE start, ITTYPE last)
{
    struct rb_node *rb = node->ITRB.rb_right, *prev;
    while (true) {
        /*
         * Loop invariants:
         *   Cond1: ITSTART(node) <= last
         *   rb == node->ITRB.rb_right
         *
         * First, search right subtree if suitable
         */
        if (rb) {
            ITSTRUCT *right = rb_entry(rb, ITSTRUCT, ITRB);
            if (start <= right->ITSUBTREE)
                return ITPREFIX ## _subtree_search(right,
                                                start, last);
        }
        /*
         * Move up the tree until we come from a node's left child */
        do {
            rb = rb_parent(&node->ITRB);
            if (!rb)
                return NULL;
            prev = &node->ITRB;
            node = rb_entry(rb, ITSTRUCT, ITRB);
            rb = node->ITRB.rb_right;
        } while (prev == rb);

        /*
         * Check if the node intersects [start;last] */
        if (last < ITSTART(node)) /* !Cond1 */
            return NULL;
        else if (start <= ITLAST(node)) /* Cond2 */
            return node;
    }
}
```

Appendix – Interval Tree

● Interval Tree kernel API

- IPREFIX에 전달된 문자열을 prefix로 가지는 아래 함수들이 생성됨
 - _iter_next
 - 현재 node의 right child에서 원하는 range 값 검색
 - 없을 경우 parent의 left child에 붙어있는 node를 발견할 때 까지 상위로 이동
 - 찾은 parent의 range 검
 - 없을 경우 위 과정 반복

```
ITSTATIC ITSTRUCT *
ITPREFIX ## _iter_next(ITSTRUCT *node, ITTYPE start, ITTYPE last) \
\\
    struct anon_vma_chain *
anon_vma_interval_tree_iter_next(struct anon_vma_chain *node,
                                  unsigned long first, unsigned long last)
{
    return __anon_vma_interval_tree_iter_next(node, first, last);
} \
    rb = node->rb_rb_right \
\\
static inline struct anon_vma_chain *
__anon_vma_interval_tree_iter_next(struct anon_vma_chain *node, unsigned long start, unsigned long last)
{
    struct rb_node *rb = node->rb_rb_right, *prev;

    while (true) {
        /*
         * Loop invariants:
         *   Cond1: avc_start_pgoff(node) <= last
         *   rb == node->rb_rb_right
         *
         * First, search right subtree if suitable
         */
        if (rb) {
            struct anon_vma_chain *right = rb_entry(rb, struct anon_vma_chain, rb);
            if (start <= right->rb_subtree_last)
                return __anon_vma_interval_tree_subtree_search(right,
                                                               start, last);
        }
        /* rmap - right 가 있으며 right의 범위 내에 해당 되면
         * right child에서 찾음 */

        /* Move up the tree until we come from a node's left child */
        do {
            /* s1 */
            rb = rb_parent(&node->rb);
            if (!rb)
                return NULL;
            prev = &node->rb;
            /* s2 */
            node = rb_entry(rb, struct anon_vma_chain, rb);
            rb = node->rb_rb_right;
        }
    }
}
```

Appendix – Interval Tree

Interval Tree kernel API

- IPREFIX에 전달된 문자열을 prefix로 가지는 아래 함수들이 생성됨
 - _iter_next
 - 현재 node의 right child에서 원하는 range 값 검색
 - 없을 경우 parent의 left child에 붙어있는 node를 발견할 때 까지 상위로 이동
 - 찾은 parent의 range 검사
 - 없을 경우 위 과정 반복

```
/*
 * rmap - 현재 node가 parent의 right child에 연결되어 있다면
 *         left child에 연결된 node가 발견될 때 까지 위로 이동
 *
 *          +   +
 *          +   +
 *          +   +
 *          L   R
 *          +   +
 *          n3  n4  n5  n6
 *          +   +
 *          n7
 *          +
 *          n8
 *          +
 *          n9
 *          +
 *          n10
 *          ...
 *
 * 즉 위 그림에서 n9에 해당하는 vm_area_struct의 rbnode 기준
 * 으로 search한다고 할 때,
 * 1. n10이하에 있는지 다 찾음.
 * 2. n1의 left child에 연결된 L까지 이동 후, L1가
 *    겹치는지 검사.
 * 3. 안 겹친다면 R기준 n6이하에 해당하는 것 있는지 검사
 * 4. 없다면 n0의 범위 검사
 * 5. ...
 *
 * case 1) prev != rb
 *         현재 node가 parent node의 left child에 연결된 경우
 *         s1           s2
 *         rb/-           -/node
 *         +
 *         +   +      =>   +   +
 *         +   +           +   *
 *         prev/node  -/-     prev/   rb/-
 *
 * case 2) prev == rb
 *         현재 node가 parent node의 right child에 연결된 경우
 *         s1           s2
 *         rb/-           -/node
 *         +
 *         +   +      =>   +   +
 *         +   +           +   *
 *         -/-  prev/node  -/-  prev/-
 *                     rb
 */
} while (prev == rb);
```

Appendix – Interval Tree

Interval Tree kernel API

- IPREFIX에 전달된 문자열을 prefix로 가지는 아래 함수들이 생성됨
 - _iter_next
 - 현재 node의 right child에서 원하는 range 값 검색
 - 없을 경우 parent의 left child에 붙어있는 node를 발견할 때 까지 상위로 이동
 - 찾은 parent의 range 검사
 - 없을 경우 위 과정 반복

```
/*
 * rmap - 현재 node 가 parent 의 right child 에 연결 되어 있다면
 *         left child 에 연결 된 node 가 발견 될 때 까지 위로 이동
 *         n0
 *         +
 *         +
 *         n1   n2
 *         +
 *         +
 *         L   R
 *         +   +
 *         n3   n4   n5   n6
 *         +
 *         n7
 *         +
 *         n8
 *         +
 *         n9
 *         +
 *         n10
 *         ...
 *
 * 즉 위 그림에서 n9에 해당하는 vm_area_struct의 rbnode 기준
 * 으로 search한다고 할 때,
 * 1. n10이하에 있는지 다 찾음.
 * 2. n1의 left child에 연결된 L까지 이동 후, L1가
 *    겹치는지 검사.
 * 3. 안 겹친다면 R기준 n6이하에 해당하는 것 있는지 검사
 * 4. 없다면 n0의 범위 검사
 * 5. ...
 *
 * case 1) prev != rb
 *         현재 node 가 parent node 의 left child 에 연결 된 경우
 * s1           s2
 *         rb/-          -/node
 *         +
 *         +   +      =>   +   +
 *         +   +          +   *
 * prev/node  -/-     prev/    rb/-
 */
/* Check if the node intersects [start;last] */
if (last < avc_start_pgoff(node)) /* !Cond1 */
    return NULL;
else if (start <= avc_last_pgoff(node)) /* Cond2 */
    return node;
/*
 * 위 그림의 L node의 범위에 나의 range가 겹치는지 검사
 * 찾았으면 OK고 못찾았으면 R에서 다시 검사 후
 */
}
while (prev == rb),
```

Reverse Mapping

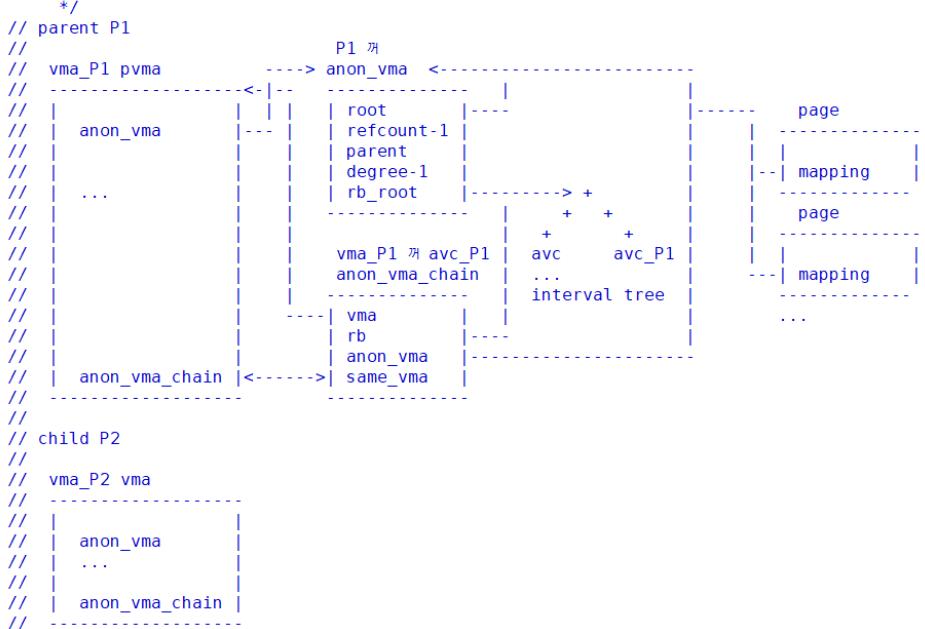
- Reverse mapping 구성 - fork
 - do_fork → copy_process → copy_mm → dup_mm → dup_mmap → anon_vma fork
 - do_fork 시 부모 process로부터 복제된 vma를 관리하기 위해 자식에도 reverse mapping 형성
 - anon_vma_fork 과정
 - 아직 자식 process의 anon_vma는 없으며 복제된 부모의 vma만 가지고 있는 상태

```

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;
    // anon_vma 관련 복제 및 tree 연결 할 수 인데 parent vma 가
    // anon_vma 가 없으면 할 일 없음

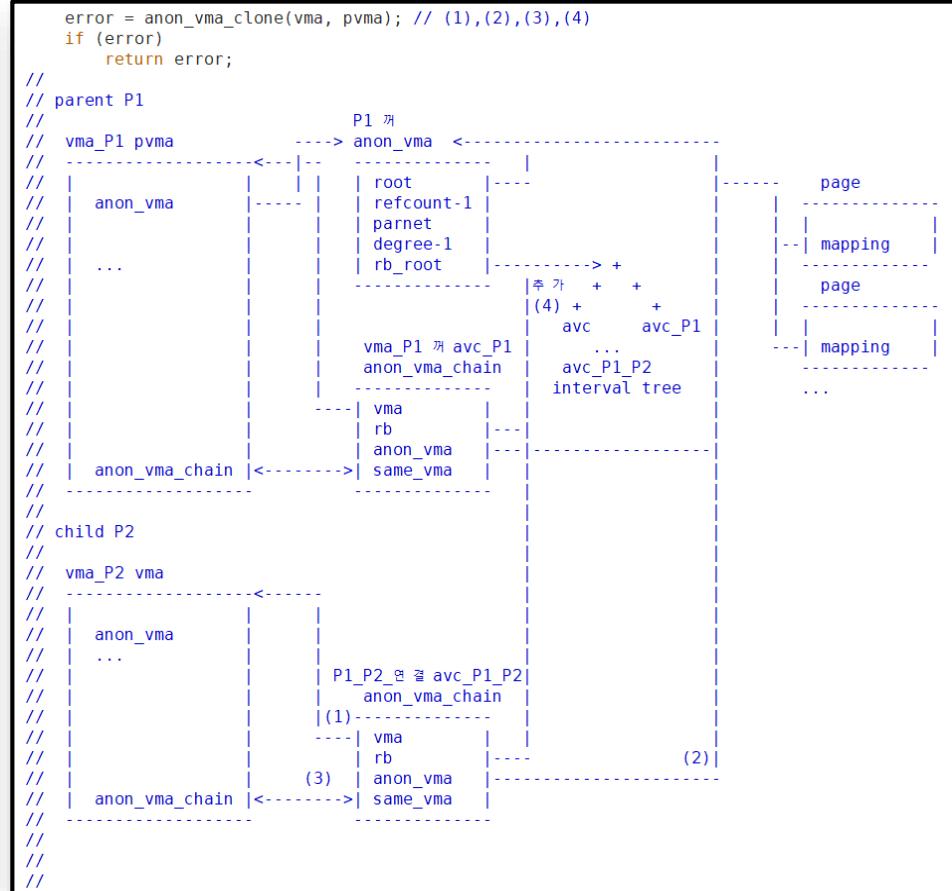
    /* Drop inherited anon_vma, we'll reuse existing or allocate new. */
    vma->anon_vma = NULL;
    // anon_vma 는 process 당 1개 쪽 존재. 따라서
    // 기존 parent 의 anon_vma 를 가리키고 있던 것을 clear 하고
    // reuse 또는 new anon_vma 사용 할 거임
    //
    /*
     * First, attach the new VMA to the parent VMA's anon_vmas,
     * so rmap can find non-COWed pages in child processes.

```



Reverse Mapping

- Reverse mapping 구성 - fork
 - do_fork → copy_process → copy_mm → dup_mm → dup_mmap → **anon_vma fork**
 - do_fork 시 부모 process로부터 복제된 vma를 관리하기 위해 자식에도 reverse mapping 형성
 - anon_vma_fork 과정
 - 아직 자식 process의 anon_vma는 없으며 복제된 부모의 vma만 가지고 있는 상태
 - anon_vma_clone 수행



Reverse Mapping

- Reverse mapping 구성 - fork
 - do_fork → copy_process → copy_mm → dup_mm → dup_mmap → anon vma fork
 - do_fork 시 부모 process로부터 복제된 vma를 관리하기 위해 자식에도 reverse mapping 형성
 - anon_vma_fork 과정
 - 아직 자식 process의 anon_vma는 없으며 복제된 부모의 vma만 가지고 있는 상태
 - anon_vma_clone 수행
 - 부모의 vma에 연결된 anon_vma_chain을 순회하며 부모와 자식의 같은 vma를 연결할 vmc 생성 및 연결

```
static void anon_vma_chain_link(struct vm_area_struct *vma,
                                struct anon_vma_chain *avc,
                                struct anon_vma *anon_vma)
{
    // fork 시 ..
    // anon_vma 는 parent process 의 anon_vma .
    // 새로 만든 복제한 vma 를 위한 avc 에 parent process 의
    // anon_vma 를 연결해 줌
    //
    avc->vma = vma; // (1)
    avc->anon_vma = anon_vma; // (2)
    list_add(&avc->same_vma, &vma->anon_vma_chain); // (3)
    // parent process 의 abc interval tree 에 COW 한 child 의
    // avc 를 추가해 줌
    // 결과적으로 요 avc 는 ...
    //     avc->vma : child 의 vma
    //     avc->rb : parent 의 interval tree root avc
    //     avc->anon_vma : parent 의 anon_vma 를 가리킴
    //     avc->same_vma : parent vma 를 복제한 child vma 를
    //                      관리하는 child 의 avc 와 연결
    //
    anon_vma_interval_tree_insert(avc, &anon_vma->rb_root); // (4)
}
```

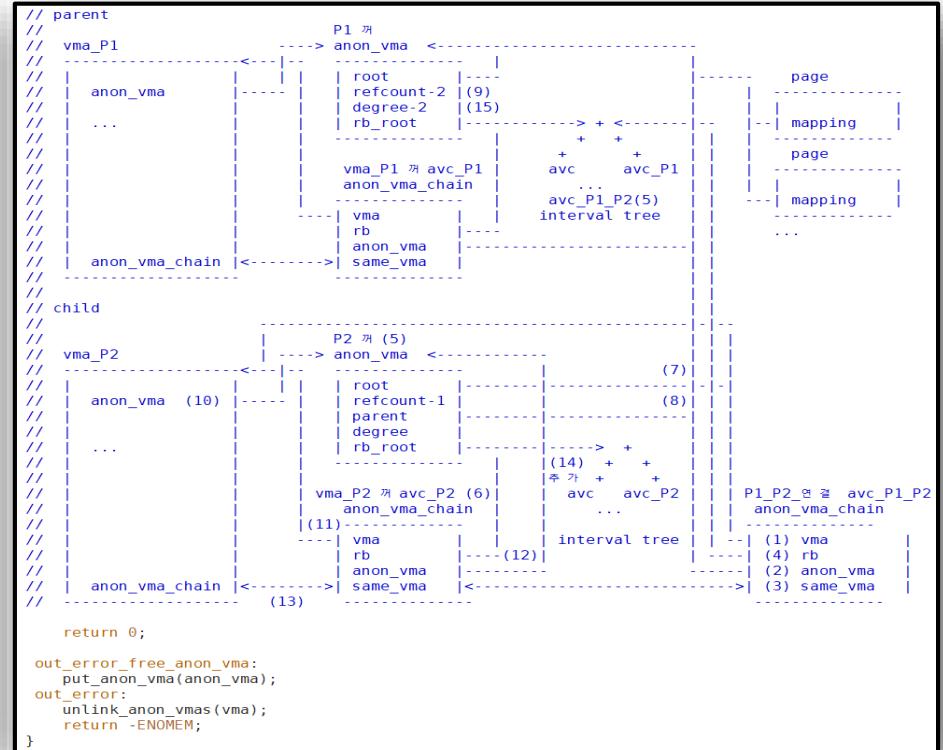
Reverse Mapping

- Reverse mapping 구성 - fork
 - do_fork → copy_process → copy_mm → dup_mm → dup_mmap → **anon_vma fork**
 - do_fork 시 부모 process로부터 복제된 vma를 관리하기 위해 자식에도 reverse mapping 형성
 - anon_vma_fork 과정
 - 아직 자식 process의 anon_vma는 없으며 복제된 부모의 vma만 가지고 있는 상태
 - anon_vma_clone 수행
 - 자식 process vma anon_vma 생성 및 현재 복제된 vma에 대한 vmc 생성하고 연결

```
/* An existing anon_vma has been reused, all done then. */
if (vma->anon_vma)
    return 0;
// anon_vma를 재사용한 경우

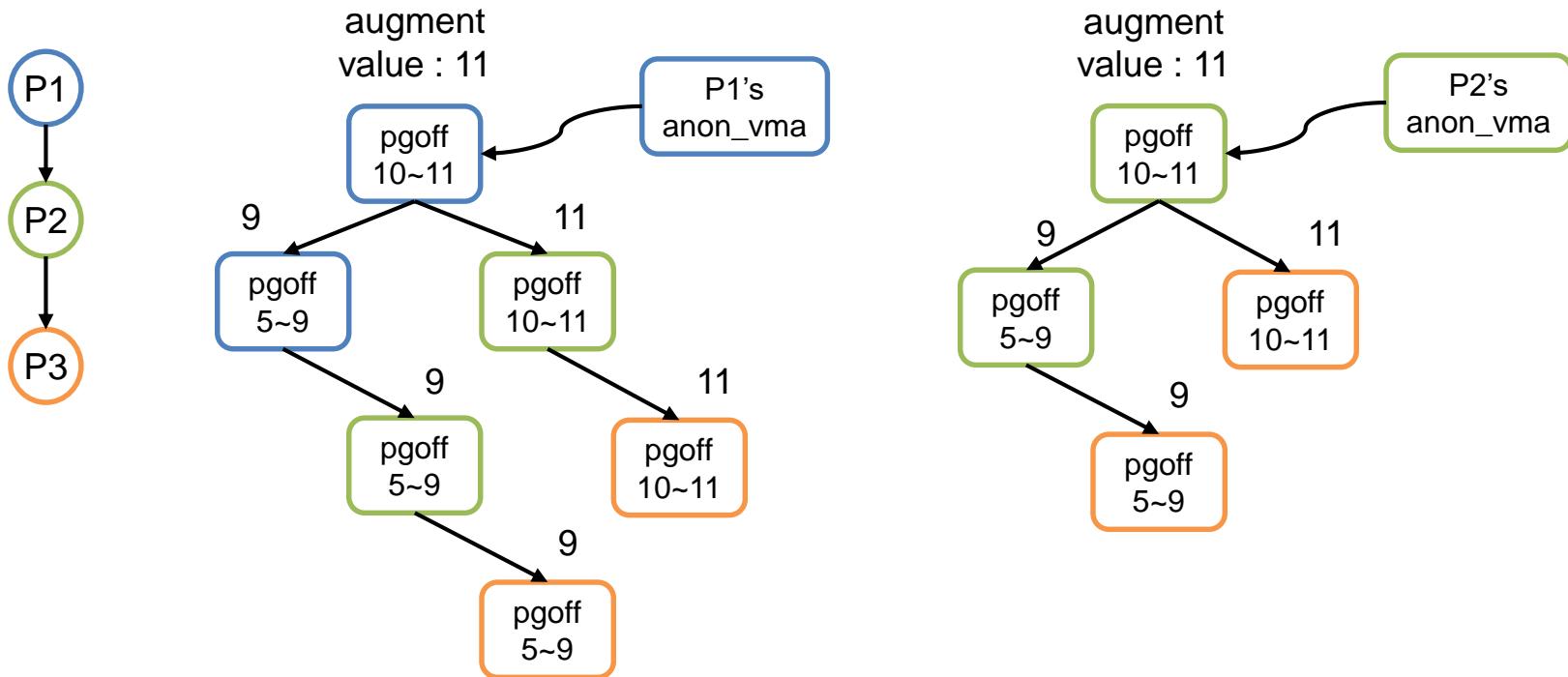
/* Then add our own anon_vma. */
anon_vma = anon_vma_alloc(); // (5)
if (!anon_vma)
    goto out_error;
avc = anon_vma_chain_alloc(GFP_KERNEL); // (6)
if (!avc)
    goto out_error_free_anon_vma;

/*
 * The root anon_vma's spinlock is the lock actually used when we
 * lock any of the anon_vmas in this anon_vma tree.
 */
anon_vma->root = pvma->anon_vma->root; // (7)
anon_vma->parent = pvma->anon_vma; // (8)
/*
 * With refcounts, an anon_vma can stay around longer than the
 * process it belongs to. The root anon_vma needs to be pinned until
 * this anon_vma is freed, because the lock lives in the root.
 */
get_anon_vma(anon_vma->root); // (9)
/* Mark this anon_vma as the one where our new (COWed) pages go. */
vma->anon_vma = anon_vma; // (10)
anon_vma_lock_write(anon_vma);
anon_vma_chain_link(vma, avc, anon_vma); // (11),(12),(13),(14)
// vma, avc, av를 연결
anon_vma->parent->degree++; // (15)
anon_vma_unlock_write(anon_vma);
```



Reverse Mapping

- Reverse mapping 구성 - fork
 - 각 process 들 중 부모 process 에 해당하는 process 들은 자신의 vma 들이 복제된 자식 vma 들에 대한 avc 들도 interval tree 에 가지고 있음
 - reverse mapping 시, next 함수를 통해 일치하는 첫번째 vma 를 찾은 후, 오른쪽 자식으로 내려가며 더 일치하는 것 있는지 검사, 더 이상 없을 시, left child 꼭대기로 이동 후 right child 부터 다시 똑같은 검사 수행



Reverse Mapping

- reverse mapping 수행 – reverse mapping
 - shrink_list → shrink_active_list → shrink_page_list
→ page_check_reference → page_referenced
 - page 를 가지고 있는 vma 들을 찾아 해당 pte 들을 수정(ACCESS BIT clear 등 수행)
 - 과정
 - 현재 page 에 map 된 vma 가 있는지 검사
 - mapping flag 되어 있는지 검사
 - lock 설정
 - reverse mapping 수행(rmap_walk)

```
// reverse mapping 을 수행 하는 함수로 page 의 anon_vma 를 통해
// anon_vma_chain 을 관리하는 interval tree 에서 page->index ~
// page->index+1 까지의 range 에 해당하는 vma 를 찾아
// _PAGE_BIT_ACCESSED 가 설정 되었을 시 , CLEAR 해 줌
int page_referenced(struct page *page,
                     int is_locked,
                     struct mem_cgroup *memcg,
                     unsigned long *vm_flags)
{
    int ret;
    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;
    // page 를 사용 중인 pte 가 없다면 종료
    if (!page_rmapping(page))
        return 0;
    // 현재 page 가 anonymous 일 수도 ... file-backed page 일 수도 ...
    // anonymous 라면 rmap 시 일단 rmap flag 인 PAGE_MAPPING_ANON
    // 등의 flag 있는지 검사
    if (!is_locked && (!PageAnon(page) || PageKsm(page))) {
        we_locked = trylock_page(page);
        if (!we_locked)
            return 1;
    }
    // Kernel shared memory 이거나 file-backed page 이며
    // lock 잡지 않은 상태로 함수가 호출된 거면 lock 잡음
    /*
     * 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;
        // cgroup 사용하는 경우 원가 함수 설정 함
    }
    ret = rmap_walk(page, &rwc);
    // reverse mapping 수행 함수 .
    // vma 찾기 , accessed bit clear
    *vm_flags = pra.vm_flags;

    if (we_locked)
        unlock_page(page);
    return pra.referenced;
}
```

조건
검사

Reverse Mapping

- reverse mapping 수행 – reverse mapping
 - shrink_list → shrink_active_list → shrink_page_list
→ page_check_reference → page_referenced
 - page 를 가지고 있는 vma 들을 찾아 해당 pte 들을 수정(ACCESS BIT clear 등 수행)
 - 과정
 - 현재 page 에 map 된 vma 가 있는지 검사
 - mapping flag 되어 있는지 검사
 - lock 설정
 - reverse mapping 수행(rmap_walk)
 - 여기서는 anonymous page 에 대해서만 스타디

```
// reverse mapping 을 수행하는 함수로 page 의 anon_vma 를 통해
// anon_vma_chain 을 관리하는 interval tree 에서 page->index ~
// page->index+1 까지의 range 에 해당하는 vma 를 찾아
// _PAGE_BIT_ACSESSED 가 설정 되었을 시 , CLEAR 해 줌
int page_referenced(struct page *page,
                     int is_locked)

// page 가 pte 에 map 되었는지 page->_mapcount 를 확인
bool page_mapped(struct page *page)
{
    int i;

    if (likely(!PageCompound(page)))
        return atomic_read(&page->_mapcount) >= 0;
    // PageCompound 즉 현재 page 가 flag 에 PG_head 설정된 head page 이거나
    // compound_head 에 head page 주소가 담긴 tail page 인지 검사 및 compound page
    // 가 아니라면 _mapcount 를 통해 현재 page 를 사용하는 pte 있는지 검사
    page = compound_head(page);
    if (atomic_read(compound_mapcount_ptr(page)) >= 0)
        return true;
    // 첫 번째 tail page 의 compound_mapcount 검사 하여 현재 compound page 가
    // map 된 pmd 가 있는지 검사
    if (PageHuge(page))
        return false;
    // THP 가 아닌 hugetlbfs 기반의 compound page 인지 검사 . hugetlbfs 기반이라면
    // 항상 어차피 memory 에 상주하도록 reserve 한 놈이므로 mapcount 가 설정 안되어
    // 있어도 괜찮
    for (i = 0; i < hpage_nr_pages(page); i++) {
        // THP 의 경우 , THP 가 pmd 에 map 된 것이 없어도 각 page 가 pte 단위로 map
        // 된 경우가 있을 수 있으므로 hugepage 크기 만큼 각 page 의 mapcount 검사
        if (atomic_read(&page[i]._mapcount) >= 0)
            return true;
    }
    return false;
}

/*
 * If we are reclaiming on behalf of a cgroup, skip
 * counting on behalf of references from different
 */
int rmap_walk(struct page *page, struct rmap_walk_control *rwc)
{
    if (unlikely(PageKsm(page)))
        return rmap_walk_ksm(page, rwc);
    else if (PageAnon(page))
        return rmap_walk_anon(page, rwc, false);
        // anonymous page 일 경우
    else
        return rmap_walk_file(page, rwc, false);
        // file backed page 일 경우
}
```

Reverse Mapping

- reverse mapping 수행 – reverse mapping
 - shrink_list → shrink_active_list → shrink_page_list
→ page_check_reference → page_referenced
 - page 를 가지고 있는 vma 들을 찾아 해당 pte 들을 수정(ACCESS BIT clear 등 수행)
 - 과정
 - 현재 page 에 map 된 vma 가 있는지 검사
 - mapping flag 되어 있는지 검사
 - lock 설정
 - reverse mapping 수행(rmap_walk)
 - LSB 에 설정된 flag 제거 후, page 의 anon_vma 를 통해 interval tree search
 - 첫번째 해당되는 vmc 찾은 후, right child 검사 및 left child 꼭대기로 이동
 - right child 로 이동

```
// anonymous page 일 경우 , process 땅 존재하는 anon_vma 를 찾아 해당 interval
// tree search 를 통해 avc 를 찾아냄
static int rmap_walk_anon(struct page *page, struct rmap_walk_control *rwc,
                           bool locked)
{
    struct anon_vma *anon_vma;
    pgoff_t pgoff_start, pgoff_end;
    struct anon_vma_chain *avc;
    int ret = SWAP AGAIN;

    if (locked) {
        anon_vma = page_anon_vma(page);
        /* anon_vma disappear under us? */
        // anonymous page 일 경우 이제 anon_vma 를 사용해야 하므로 LSB flag 제거
        VM_BUG_ON_PAGE(!anon_vma, page);
    } else {
        anon_vma = rmap_walk_anon_lock(page, rwc);
    }
    if (!anon_vma)
        return ret;
    // reverse mapping 에서의 검색을 위해 virtual address 에서의 start,end 가져옴
    pgoff_start = page_to_pgoff(page);
    // virtual address 에서의 start offset 즉 insex 가져옴
    pgoff_end = pgoff_start + hpage_nr_pages(page) - 1;
    // index 를 통해 huge page 의 경우 , 512 개로 , 일반 page 의 경우 1 개로
    // virtual page offset end 를 계산
    anon_vma_interval_tree_foreach(avc, &anon_vma->rb_root,
                                    pgoff_start, pgoff_end) {
        // anon_vma 의 rbtree 에 존재하는 anon_vma_chain 을 순회
        // 계산한 start-end 까지의 범위를 가지는 avc 를 anon_vma 가 가리키는
        // interval tree 에서 찾음
        // for (avc = anon_vma_interval_tree_iter_first(root, start, last);
        //      avc;
        //      avc = anon_vma_interval_tree_iter_next(avc, start, last))
        //
        // __anon_vma_interval_tree_iter_first(struct rb_root *root,
        //                                     unsigned long start, unsigned long last)
        // : 일단 start<rb_subtree_last 를 통해 범위 내에 존재하는지
        // 판단 후 , search 수행
        // __anon_vma_interval_tree_iter_next(struct anon_vma_chain *node,
        //                                     unsigned long start, unsigned long last)
        // : 찾은 avc 기준으로 parent 탐색서 left child 연결된 것 찾아
        //      right 에서 다시 검사
        struct vm_area_struct *vma = avc->vma;
        // internal tree 에서 search 결과 찾은 vmc 가 관리하는 vma 가져옴
        unsigned long address = vma_address(page, vma);
        // vm_start 를 가져온다고 생각해도 될듯 .
        cond_resched();

        if (rwc->invalid_vma && rwc->invalid_vma(vma, rwc->arg))
            continue;
        // cgroup 이 설정되었을 경우에만 호출 및 검사

        ret = rmap_one(page, vma, address, rwc->arg);
        if (ret != SWAP AGAIN)
            break;
        if (rwc->done && rwc->done(page))
            break;
    }

    if (!locked)
        anon_vma_unlock_read(anon_vma);
    return ret;
}
```

Reverse Mapping

- reverse mapping 수행 – reverse mapping
 - shrink_list → shrink_active_list → shrink_page_list
→ page_check_reference → page_referenced
 - page 를 가지고 있는 vma 들을 찾아 해당 pte 들을 수정(ACCESS BIT clear 등 수행)
 - 과정
 - 현재 page 에 map 된 vma 가 있는지 검사
 - mapping flag 되어 있는지 검사
 - lock 설정
 - reverse mapping 수행(rmap_walk)
 - LSB 에 설정된 flag 제거 후, page 의 anon_vma 를 통해 interval tree search
 - 첫번째 해당되는 vmc 찾은 후, right child 검사 및 left child 꼭대기로 이동
 - right child 로 이동
 - 찾은 vmc 의 vma 에 대하여 rmap_one 수행

```
// anonymous page 일 경우 , process 랑 존재 하는 anon_vma 를 찾아 해당 interval
// tree search 를 통해 avc 를 찾아냄
static int rmap_walk_anon(struct page *page, struct rmap_walk_control *rwc,
                           bool locked)
{
    struct anon_vma *anon_vma;

    struct page_vma_mapped_walk {
        struct page *page;
        // rmap 에서 확인 할 page
        // frame 에 해당하는 page
        struct vm_area_struct *vma;
        // Interval tree 에서 찾은
        // avc 가 가진 vma 로 이 vma
        // 의 pte 찾아 접근
        unsigned long address;
        // pte 찾을 virtual address
        pmd_t *pmd;
        // pmd 주소
        pte_t *pte;
        // pte 주소
        spinlock_t *ptl;
        // pte 접근 시 사용할 lock
        unsigned int flags;
    };

    // interval tree에서 찾음
    // for (avc = anon_vma_interval_tree_iter_first(root, start, last);
    //      avc;
    //      avc = anon_vma_interval_tree_iter_next(avc, start, last))
    //      __anon_vma_interval_tree_iter_first(struct rb_root *root,
    //                                         unsigned long start, unsigned long last)
    //      : 일단 start<rb_subtree_last 를 통해 범위 내에 존재하는지
    //      판斷 후, search 수행
    //      __anon_vma_interval_tree_iter_next(struct anon_vma_chain *node,
    //                                         unsigned long start, unsigned long last)
    //      : 찾은 avc 기준으로 parent 탐색해 left child 연결된 것 찾음
    //      right에서 다시 검사
    struct vm_area_struct *vma = avc->vma;
    // internal tree에서 search 결과 찾은 vmc 가 관리하는 vma 가져옴
    unsigned long address = vma_address(page, vma);
    // vm_start 를 가져온다고 생각해도 될듯.
    cond_resched();

    if (rwc->invalid_vma && rwc->invalid_vma(vma, rwc->arg))
        continue;
    // cgroup 이 설정되었을 경우에만 호출 및 검사

    ret = rwc->rmap_one(page, vma, address, rwc->arg);
    if (ret != SWAP AGAIN)
        break;
    if (rwc->done && rwc->done(page))
        break;
}

if (!locked)
    anon_vma_unlock_read(anon_vma);
return ret;
}
```

virtual address에서의 start,end 가져옴
ret 즉 insex 가져옴
pages(page) - 1;
512개로, 일반 page의 경우 1개로
&anon_vma->rb_root,
anon_vma_chain을 순회
가지는 avc를 anon_vma가 가리키는

struct rmap_walk_control rwc = {
.rmap_one = page_referenced_one,
.arg = (void *)&ra,
.anon_lock = page_lock_anon_vma_read,

Reverse Mapping

- reverse mapping 수행 – reverse mapping
 - shrink_list → shrink_active_list → shrink_page_list
→ page_check_reference → page_referenced
 - page 를 가지고 있는 vma 들을 찾아 해당 pte 들을 수정(ACCESS BIT clear 등 수행)
 - 과정
 - 현재 page 에 map 된 vma 가 있는지 검사
 - mapping flag 되어 있는지 검사
 - lock 설정
 - reverse mapping 수행(rmap_walk)
 - LSB 에 설정된 flag 제거 후, page 의 anon_vma 를 통해 interval tree search
 - 첫번째 해당되는 vmc 찾은 후, right child 검사 및 left child 꼭대기로 이동
 - right child 로 이동
 - 찾은 vmc 의 vma 에 대하여 rmap_one 수행
 - ACCESSED BIT clear 수행

```
// interval tree에서 찾아온 vma에 대하여 pmd, pte.. 등을 구해 accessed bit clear
static int 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)) {
        // pvmw.address를 통해 pmd, pte를 가져오는 등의 일 수행
        address = pvmw.address;

        if (vma->vm_flags & VM_LOCKED) {
            page_vma_mapped_walk_done(&pvmw);
            pra->vm_flags |= VM_LOCKED;
            return SWAP_FAIL; /* To break the loop */
        }
        // VM_LOCKED 이면 page out 되면 안되므로 fail
        if (pvmw.pte) {
            if (pte_clear_flush_young_notify(vma, address,
                                              pvmw.pte)) {
                // 설정한 pvmw.pte에 _PAGE_BIT_ACCESSED 가 설정되어 있다면
                // 즉 결과적으로
                // _PAGE_BIT_PRESENT 가 있어서 현재 memory에 있는 상태고
                // _PAGE_BIT_ACCESSED 가 설정되어 있어서 최종적으로 해당 pte에
                // physical address가 mapping된 것을 확인했다면 clear 수행
                /*
                 * 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_HUGE PAGE)) {
            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 SWAP_SUCCESS; /* To break the loop */

    return SWAP AGAIN;
}
```

Q & A