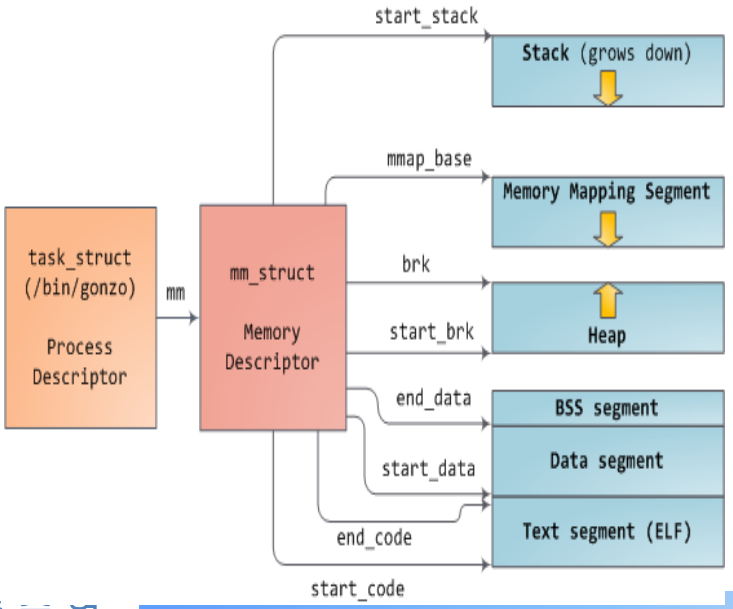
# Virtual Process Memory

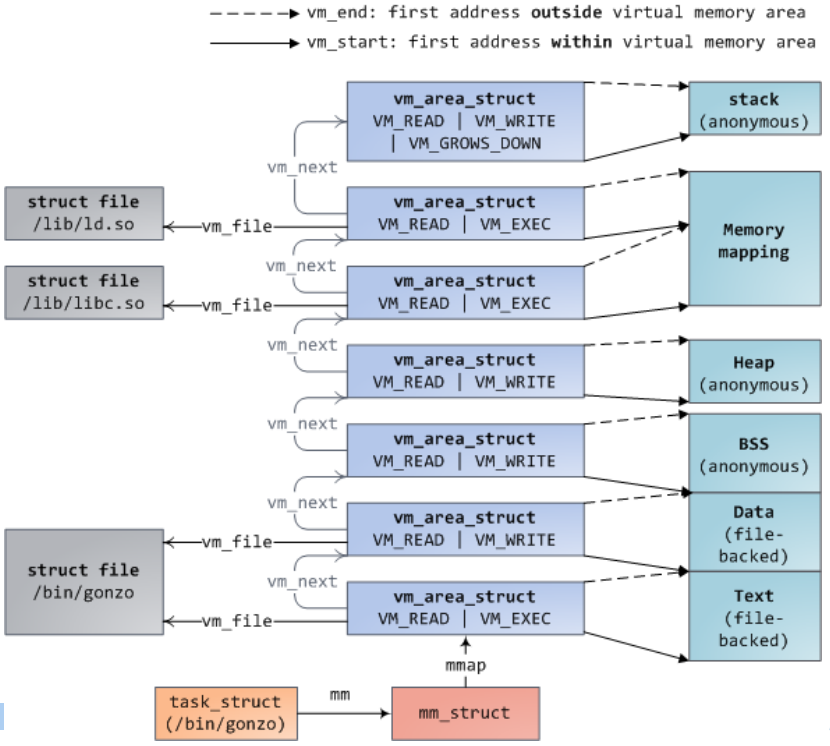
Son Ju Hyung

tooson9010@gmail.com

# Virtual Process Memory Overview

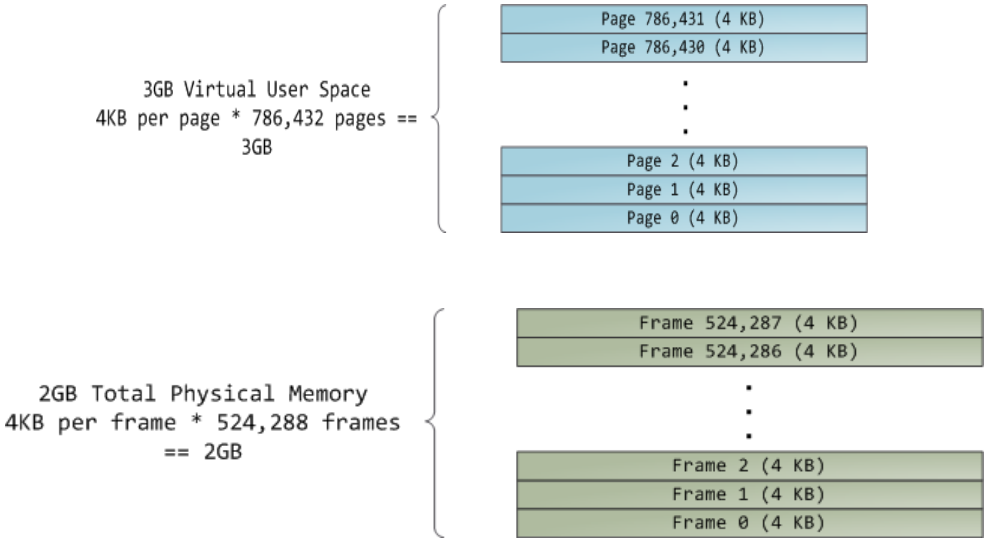
- linux 는 task_struct 를 통해 process 를 관리하며 task_struct->mm 을 통해 process 가 사용하는 memory 를 관리
  - mm 내부에 각 segment 들의 시작 끝 주소 관리
- process 가 접근하는 virtual memory area 는 vm_area_struct 를 통해 관리
  - 각 vma 는 연속적 공간이며 mmap 을 제외하고 mmap 영역을 제외하고 하나씩 존재(x86) 하며 virtual area 의 시작, 끝을 나타냄
  - starting virtual address 부터 single linked list 로 연결된 구조로 이루어져 있으며 빠른 검색 을 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



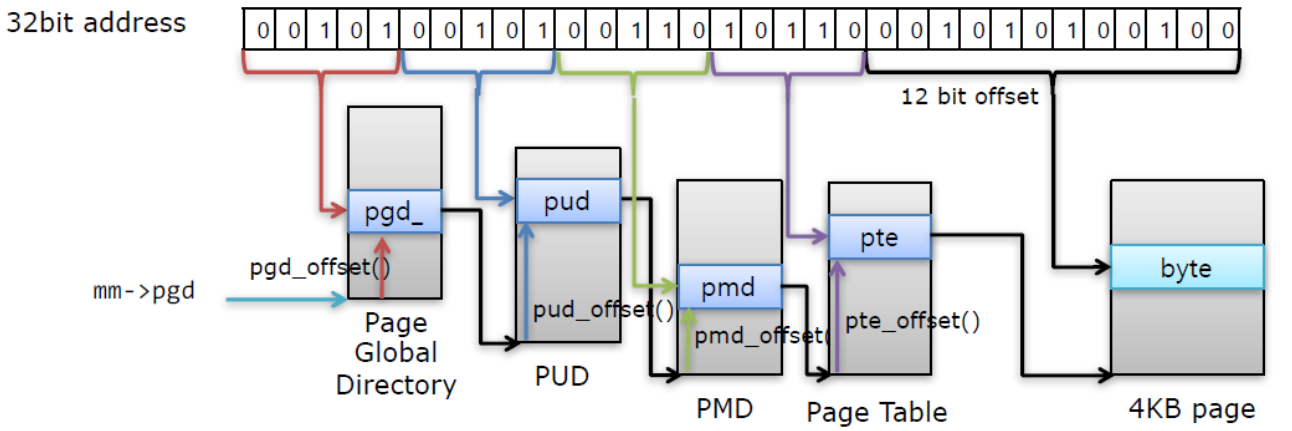
# Virtual Process Memory Overview

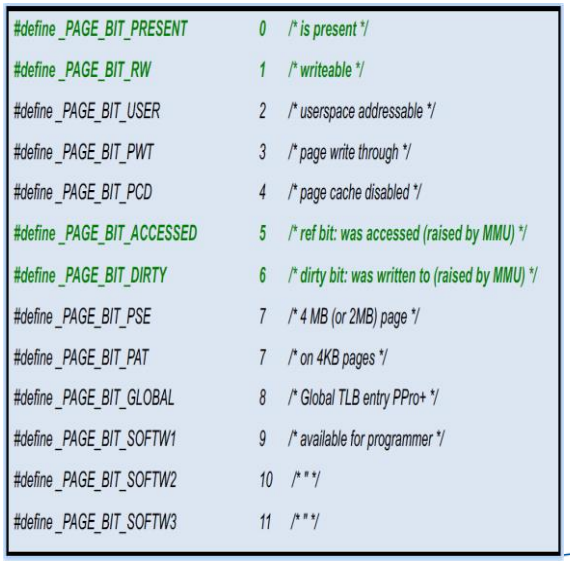
- linux 는 task_struct 를 통해 process 를 관리하며 task_struct->mm 을 통해 process 가 사용하는 memory 를 관리
  - mm 내부에 각 segment 들의 시작 끝 주소 관리
- process 가 접근하는 virtual memory area 는 vm_area_struct 를 통해 관리
  - 각 vma 는 연속적 공간이며 mmap 을 제외하고 mmap 영역을 제외하고 하나씩 존재(x86) 하며 virtual area 의 시작, 끝을 나타냄
  - starting virtual address 부터 single linked list 로 연결된 구조로 이루어져 있으며 빠른 검색 을 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 지게 되는 주소는 4KB 단위로 관리



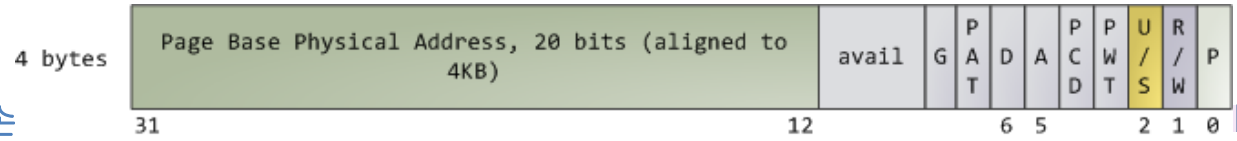
# Virtual Process Memory Overview

- 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 다름을 의미하는 flag, write-back 에 사용
  - A(PAGE_ACCESSED) : page reclaim 시 reverse mapping 에 사용



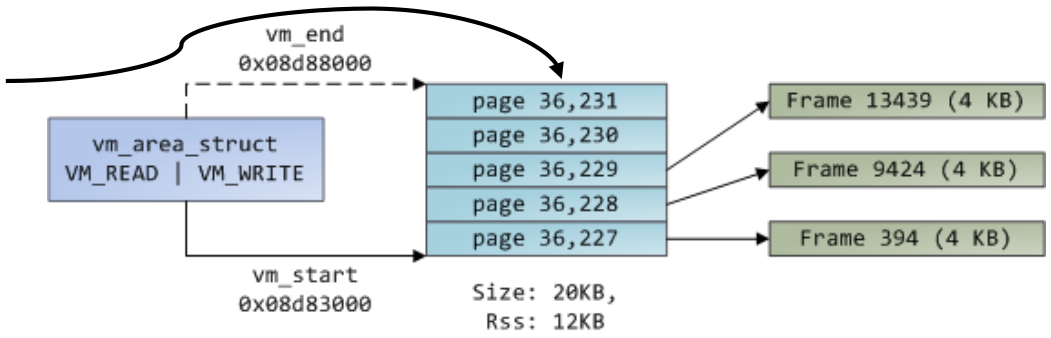
```
#define _PAGE_BIT_PRESENT      0   /* is present */
#define _PAGE_BIT_RW           1   /* writeable */
#define _PAGE_BIT_USER         2   /* userspace addressable */
#define _PAGE_BIT_PWT          3   /* page write through */
#define _PAGE_BIT_PCD          4   /* page cache disabled */
#define _PAGE_BIT_ACCESSED     5   /* ref bit: was accessed (raised by MMU) */
#define _PAGE_BIT_DIRTY        6   /* dirty bit: was written to (raised by MMU) */
#define _PAGE_BIT_PSE          7   /* 4 MB (or 2MB) page */
#define _PAGE_BIT_PAT          7   /* on 4KB pages */
#define _PAGE_BIT_GLOBAL       8   /* Global TLB entry PPro+ */
#define _PAGE_BIT_SOFTW1       9   /* available for programmer */
#define _PAGE_BIT_SOFTW2       10  /* " */
#define _PAGE_BIT_SOFTW3       11  /* " */
```



# Virtual Process Memory Overview

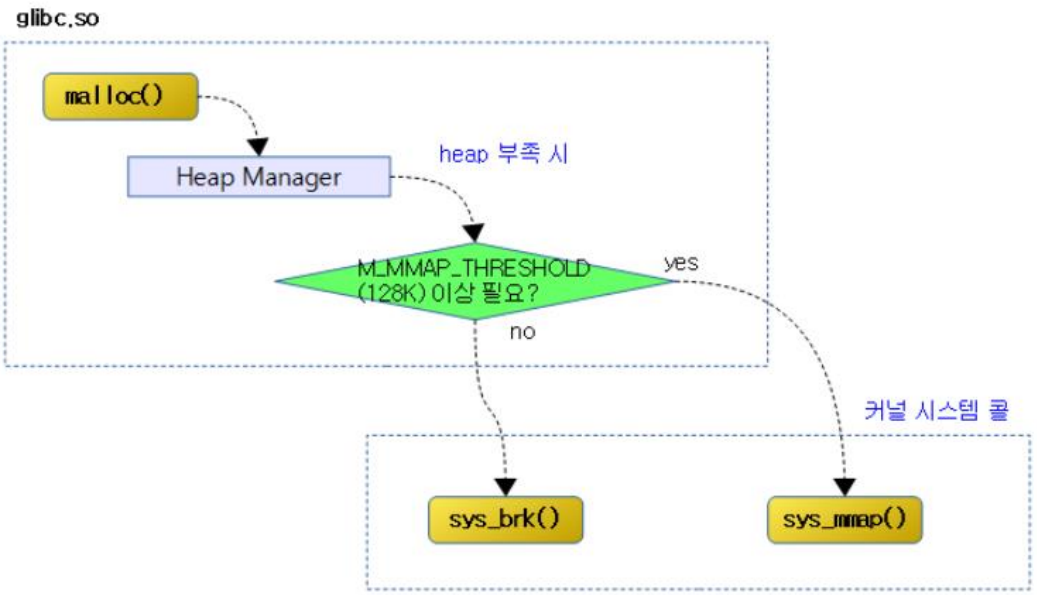
- vma 를 통해 나타내어지는 page 의 실제 할당된 위치는 page table 을 통해 virtual address 에 대한 physical memory address(4KB 단위 align) 를 관리, process 마다 각자의 page table 을 가짐
- page talbe 의 각 entry (PTE) 에는 physical address 이외에도 해당 물리 page frame 의 여러가지 상태를 기술하는 attribute 가 있음
  - P(PAGE_PRESENT) : virtual page에 대한 physical page 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 다름을 의미하는 flag, write-back 에 사용
  - A(PAGE_ACCESSED) : page reclaim 시 reverse mapping 에 사용

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



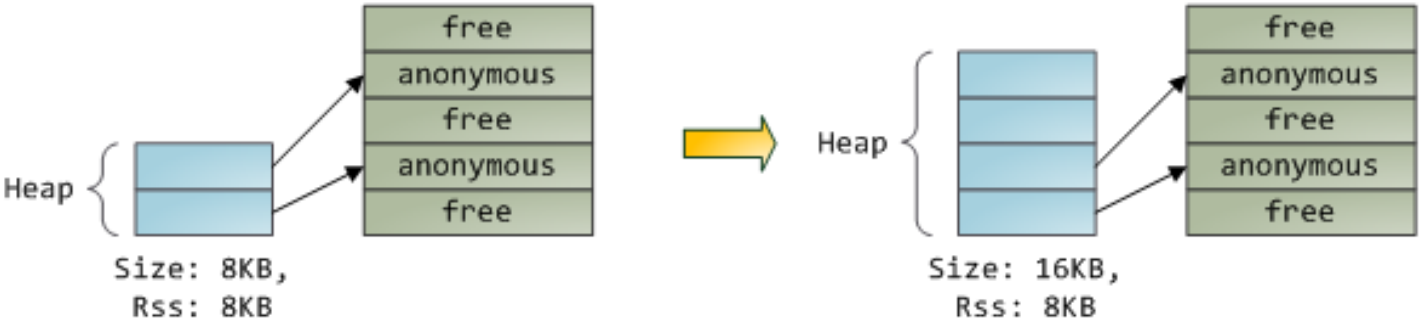
# Virtual Process Memory Overview

- 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.
  1. 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 할당 하거나 sys_mmap 을 통해 새로운 vma 생성



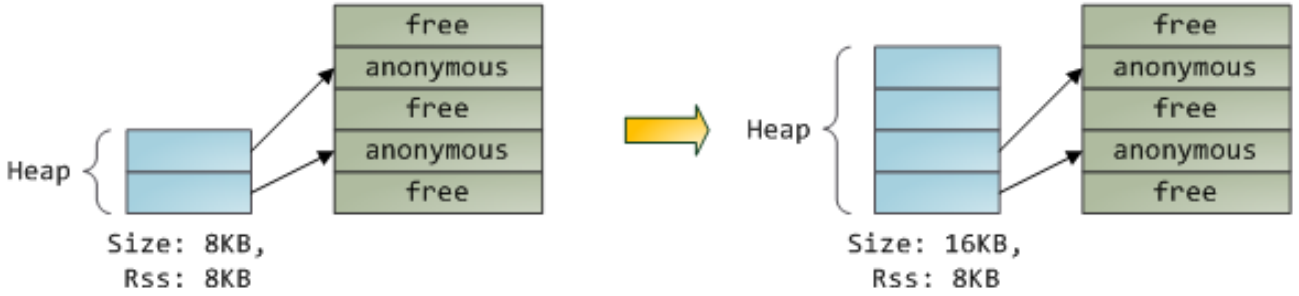
# Virtual Process Memory Overview

- 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.
  1. malloc() 을 통해 memory 할당 (anonymous page, 8KB 할당 요청 가정)
  2. 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

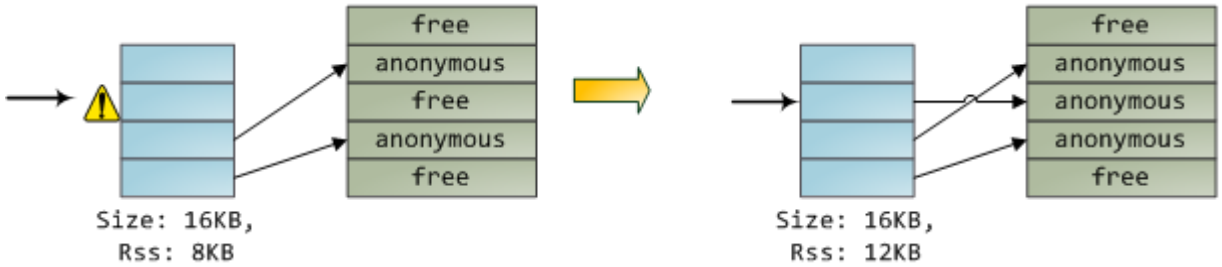


# Virtual Process Memory Overview

- 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.
  1. malloc() 을 통해 memory 할당 (anonymous page, 8KB 할당 요청 가정)
  2. heap 영역에 해당하는 vma 영역의 크기 증가 및 mm update 등 수행

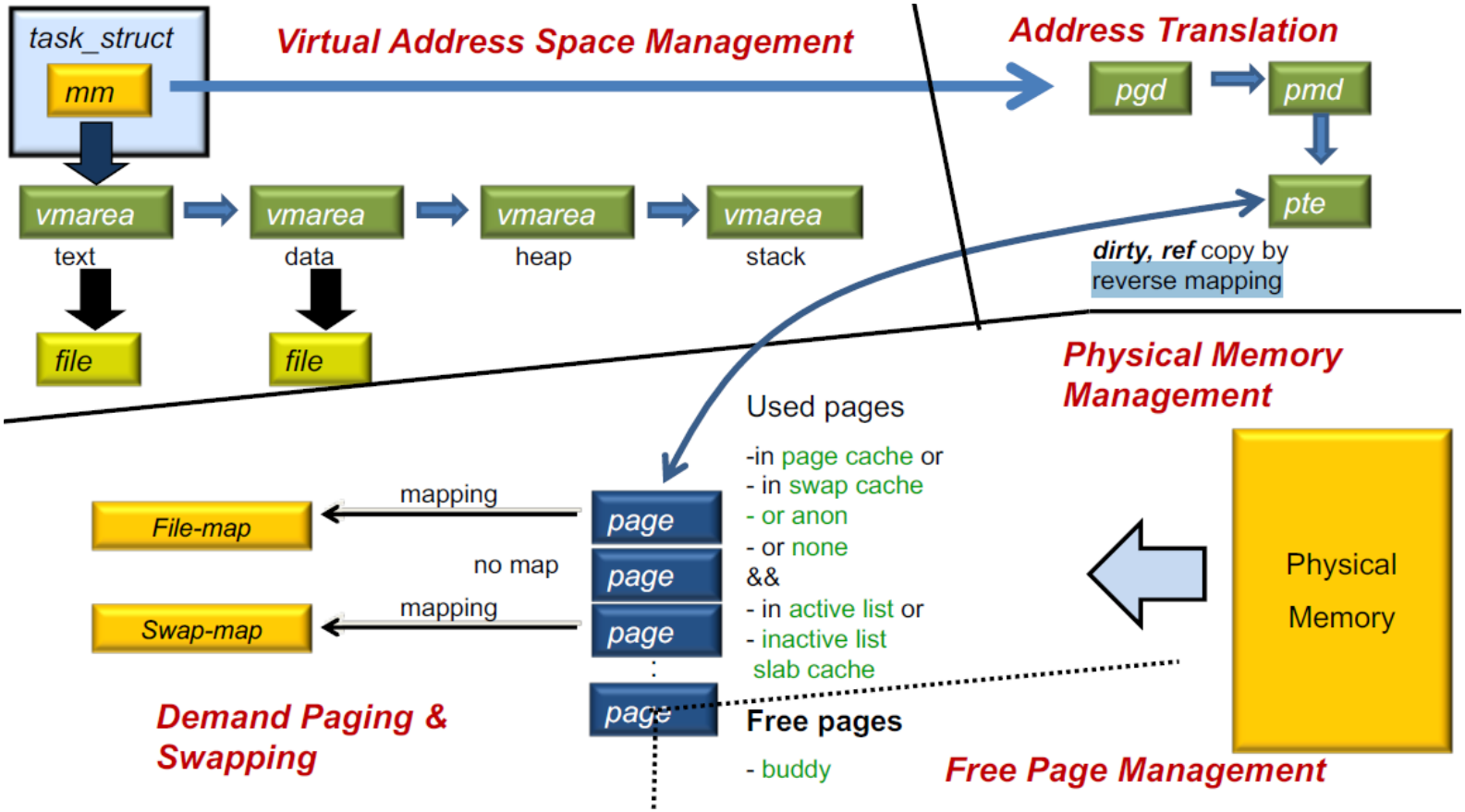



1. 추가된 virtual address 영역의 page 에 접근 시, page fault 발생
   - do_page_fault
     - find_vma
     - pte attribute 를 확인하여 PAGE_PRESENT bit clear 확인 및 physical frame 주소 clear 확인
     - do_anonuymous_page 를 통해 사용가능 한 physical frame 가져와 pte 및 vma 와 연결
     - ❖ PAGE_PRESET bit 가 clear 되어 있으나, physical frame 주소가 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 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 )의   I
    // vmacache 에 해당하는 sequence number
#ifdef 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 larget 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 사용하는 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 감소하며 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 liking pointer

    struct rb_node vm_rb;
    // red black tree 에서의 현재 vm_area_struct 가 속한 node 로
    // rb tree 에서 찾아서 container of 로 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 그전 vma 들 사이에서의 gap 들
    // 중 가장 큰 gap 으로 get_unmapped_area 에서 현재 vma 들 사이에 낄 수 있긴
    // 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 을 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 설정될수 있다 ?(mprotect)
    //       VM_GROWSDOWN, VM_GROWSUP : stack 은 VM_GROWSDOWN, heapd 은 VM_GROWSUP
    //       VM_DONTCOPY : fork 시 해당 vm 영역을 copy 하지 말것
    //       VM_DONTEXPAND : vm 영역 rmremap 등으로 확장 불가능
```

# Data Structures

- struct vm_area_struct

```
    struct {
        struct rb_node rb;
        // left subtree 들 중 에 서 의  vm_end max 값 을  가 지 는  node
        unsigned long rb_subtree_last;
        // left subtree 들 중 에 서 의  vm_end max 값
    } shared;
    // 옛 날 엔  prio_tree 로  관 리  되 었 지 만 , rb_tree 로  관 리 되 도 록  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 되 기  전 의  dirty anon, dirty data 관 리

    struct anon_vma *anon_vma;  /* Serialized by page_table_lock */
    // anonymous page COW handling, shared page 관 리
    // reverse mapping 관 련

    /* Function pointers to deal with this struct. */
    const struct vm_operations_struct *vm_ops;
    // demand paging 을  위 한  open, close, mmap 함 수 등

    /* Information about our backing store: */
    unsigned long vm_pgoff;     /* Offset (within vm_file) in PAGE_SIZE units */
    // file mapping 시 에  전 체  file 이  mapping 된  것 이 라 면  0 으 로 , 부 분 만  mapping
    // 된  것 이 라 면  그 offset 을  의 미 (page 수  기 준 )
    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
#ifdef CONFIG_NUMA
    struct mempolicy *vm_policy;    /* NUMA policy for the VMA */
#endif
    struct vm_userfaultfd_ctx vm_userfaultfd_ctx;
};
```

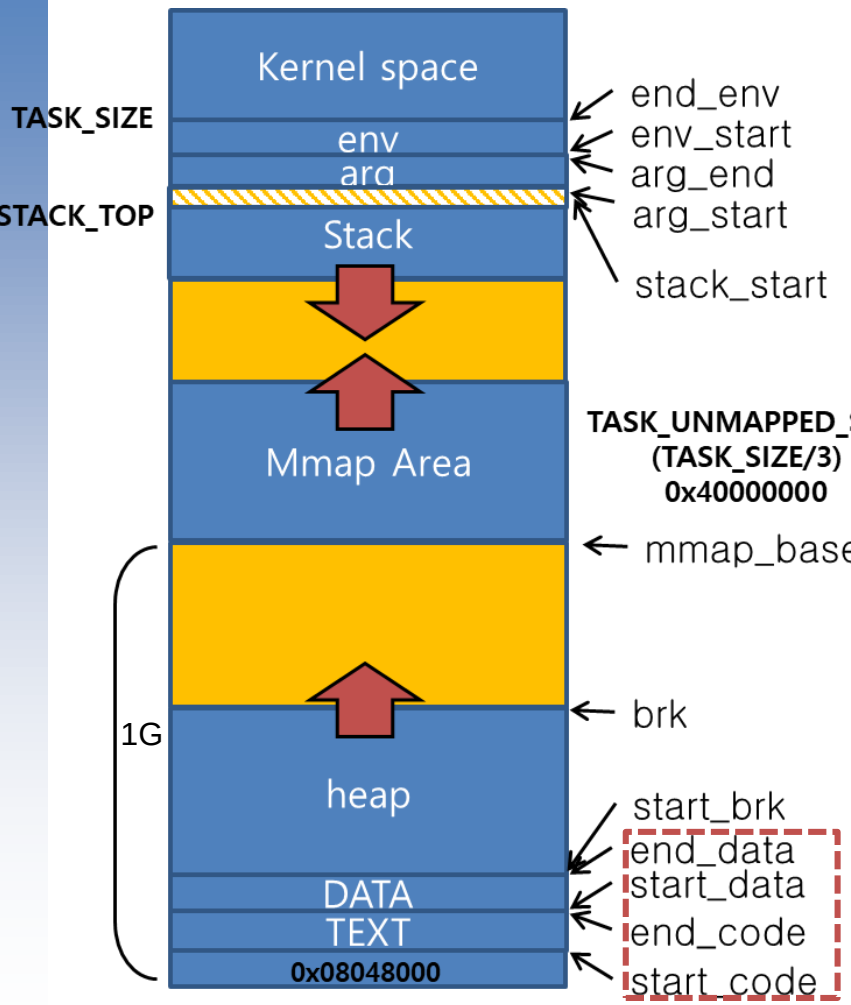
# Data Structures

- 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

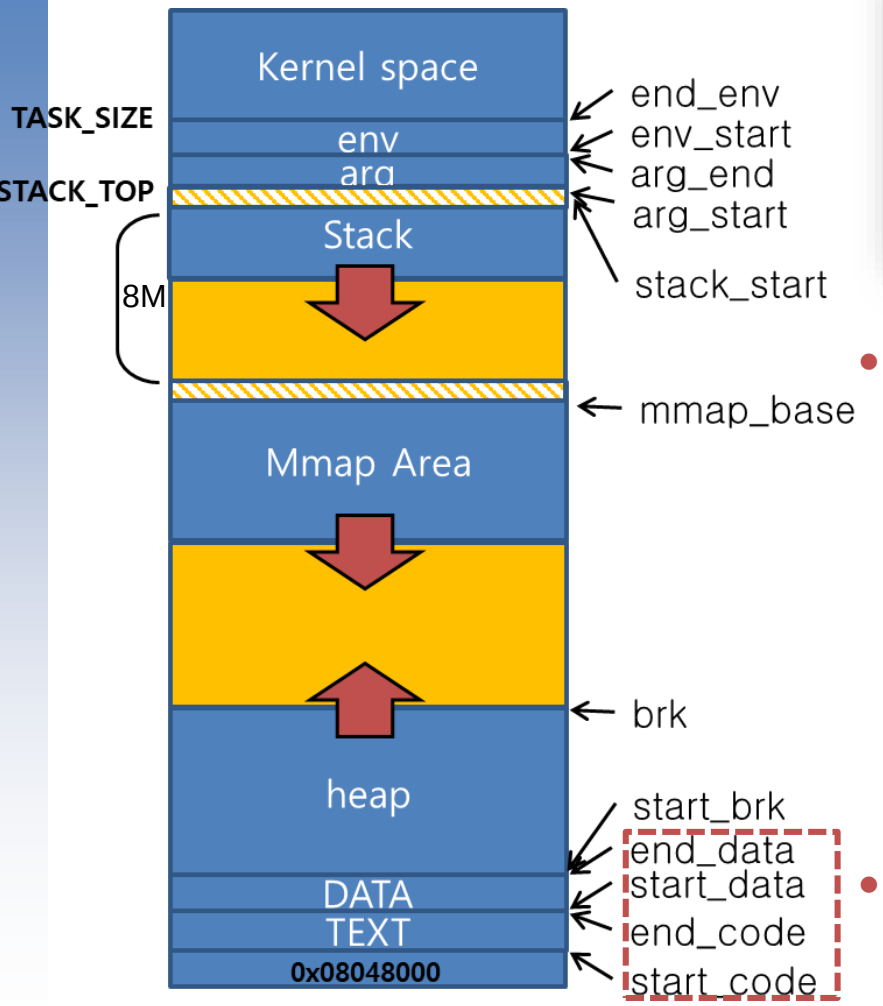
# Virtual Process Address Space Layout




```
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 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 : envvironment 영역  start ~ end address
```

- Classical virtual memory layout
  - mm_struct 를 통해 virtual address apace 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)

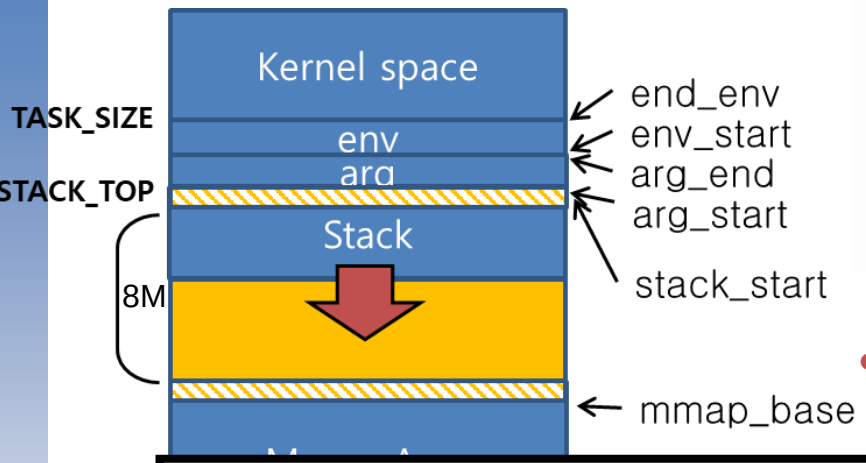
# Virtual Process Address Space Layout




```
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 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 : envvironment 영역  start ~ end address
```

- Classical Virtual memory layout
  - mm_struct 를 통해 virtual address apace 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 : envvironment 영역  start ~ end address
```

- Classical Virtual memory layout
  - mm_struct 를 통해 virtual address apace 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              unlimited            bytes
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
```

8MB

# 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/ransomize_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 을 통 해  randomize 하 지  말 도 록  설 정 한  상 태 가
    // 아 니 고 ,randomize_va_space 가  sysfs 를  통 해  설 정 되 어  있 다 면  VA layout
    // 생 성 시  randomize 하 도 록  task_struct 에  flag 설 정 (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);
    // 지 정 된  위 치 에  stack 을  생 성 하 며  arch specific 한  STACK_TOP 을  넘 겨 주 며 ,
    // PF RANDOMIZE 설 정 시 , top 위 치 에  대 해  randomize 수 행


    ...

}
```

```
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/ransomize_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

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

```
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 생성하며 arch specific 한 STACK_TOP 을 넘겨주며 ,
    // PF RANDOMIZE 설정시 , top 위치에 대해 randomize 수행

    ...
}
```

```
// new process 의 vma layout 생성시마다 호출되어 layout type 결정
void arch_pick_mmap_layout(struct mm_struct *mm)
{
    unsigned long random_factor = 0UL;
    // PF_RANDOMIZE 설정되어 있다면 즉 vma 에 randomize 적용할 꺼면
    // 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 고정위치
        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 설정된 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, 등을 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
```

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

```
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 생성하며 arch specific 한 STACK_TOP 을 넘겨주며,
    // PF_RANDOMIZE 설정시, top 위치에 대해 randomize 수행

    ...

}
```

```
// new process 의 vma layout 생성시마다 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 설정되어 있는지 검사
        if (rlimit(RLIMIT_STACK) == RLIM_INFINITY)
            return 1;
        // tsk->signal->rlim[limit].rlim_cur 에 설정된 rlimit
        // 배열요소 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 침범하지 않기 위해
        // stack size 가 고정이기 때문에 RLIM_INFINITY 로 설정되어 있다면
        // legacy layout 임
        return sysctl_legacy_va_layout;
        // /proc/sys/vm/legacy_va_layout 에 설정된 값을 반환
    }
    ...으로
        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 설정된 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

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

```
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 생성하며 arch specific 한 STACK_TOP 을 넘겨주며 ,
    // PF RANDOMIZE 설정시 , top 위치에 대해 randomize 수행

    ...
}
```

```
// new process 의 vma layout 생성시마다 호출되어 layout type 결정
void arch_pick_mmap_layout(struct mm_struct *mm)
{
    unsigned long random_factor = 0UL;
    // PF_RANDOMIZE 설정되어 있다면 즉 vma 에 randomize 적용할 꺼면
    // 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 고정위치
        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 설정된 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

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

```
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 생성하며 arch specific 한 STACK_TOP 을 넘겨주며 ,
    // PF RANDOMIZE 설정시 , top 위치에 대해 randomize 수행

    ...
}
```

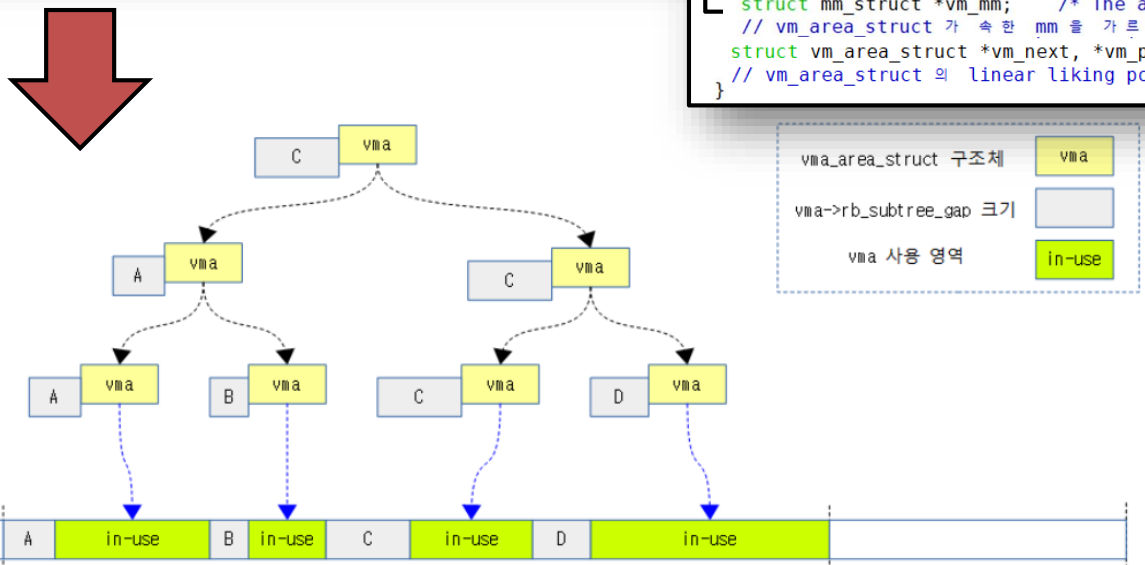
```
static unsigned long mmap_base(unsigned long rnd)
{
    unsigned long gap = rlimit(RLIMIT_STACK);
    // default layout 일 경우 , stack size 가 고정임
    if (gap < MIN_GAP)
        gap = MIN_GAP;
    // MIN_GAP : 128*1024*1024UL + stack_maxrandom_size()
    //           stack 크기가 최소 128 MB 는 되도록 설정
    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
    //  | rnd v | -> rnd         +
    //  | stack | +              |
    //  |       | |  stack size  | gap
    //  |       | |              |
    //  |       | +              |
    //  |-------|                +
    //  | rnd v | -> rnd
    //  ---------
    //  | mmap  |
    //   ...
    //
}
```

# 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.

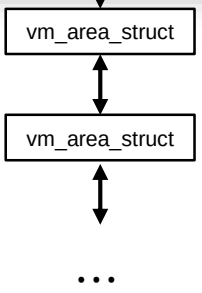
```
struct mm_struct {
    struct vm_area_struct *mmap;        /* list of VMAs */
    // mm 이 가진 vm area 의 정보를 나타내는 single linked list
    struct rb_root mm_rb;
    // vm_area_struct 와 관련된 rb- tree 로 rb tree 의 root 를 가리킴
    u32 vmacache_seqnum;                /* per-thread vmacache */
    // task_struct 별로 가지고 있는 VMCACHE_SIZE 크기 (vm_area_struct 4개 )의
    // 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 그전 vma 들 사이에서의 gap 들
  // 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 을 가르키기 위한 back-pointer
 struct vm_area_struct *vm_next, *vm_prev;
 // vm_area_struct 의 linear liking pointer
}
```



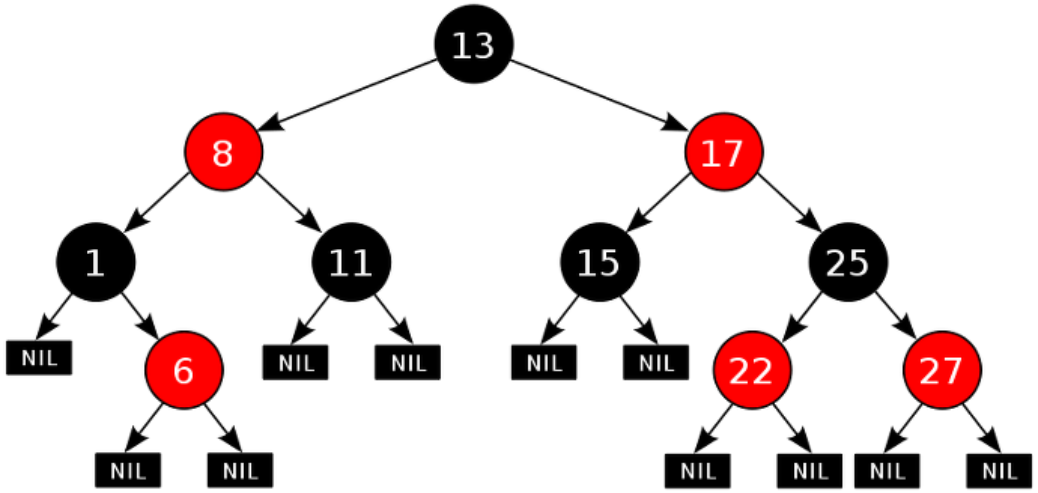






# Appendix – Red Black Tree

- 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 tree 이며 최악의 경우에도 log(N) 의 시간에 검색/삽입/삭제 수행anticipatory, deadline, CFQ, I/O scheduler, vma 관리, reverse mapping 등에서 사용
  - linux 에서 rbtree 를 사용하는 곳이 많으며 각각의 필요에 따라 비교하는 알고리즘 또한 다르기 때문에 insert/search 함수 구현 및 locking 을 개발자에게 넘김
  - Rule
    1. 모든 node 는 red 또는 black
    2. root node 는 black, new node 는 red
    3. red node 의 자식이 red 가 될 수는 없음. 다른 경우는 다 가능
    4. root 에서 leaf 까지의 black node 의 개수는 동일

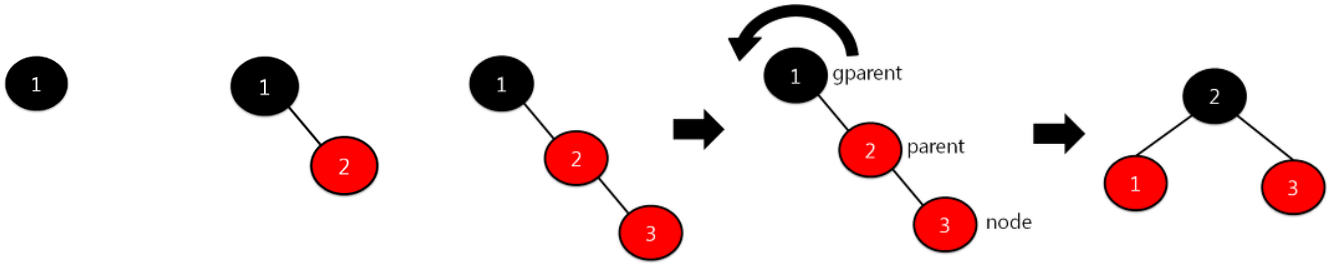


# Appendix – Red Black Tree

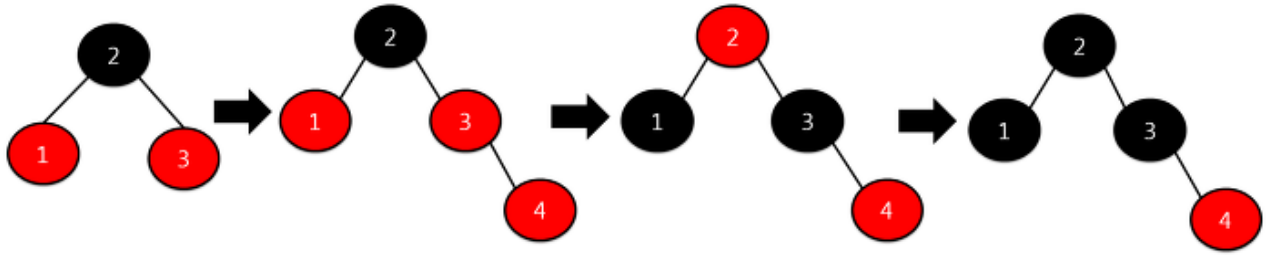
- classic red-black tree case by Rule 4.
  - case by Rule 4. – red node 는 red node 를 자식으로 가질 수 없음
    1. uncle 없고, new 가 parent 의 left child
       1. 2 와 같으며 방향 반대
    2. uncle 없고, new 가 parent 의 right child
       1. gparent 위치 기준으로 right rotate
       2. parent 위치에서 child 위치로 내려가는 놈은 red 로 설정 & child 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 내려가게된 gparent 위치 였던 node 의 right sub tree 로 입양
       4. Rule 1 검사

# Appendix – Red Black Tree

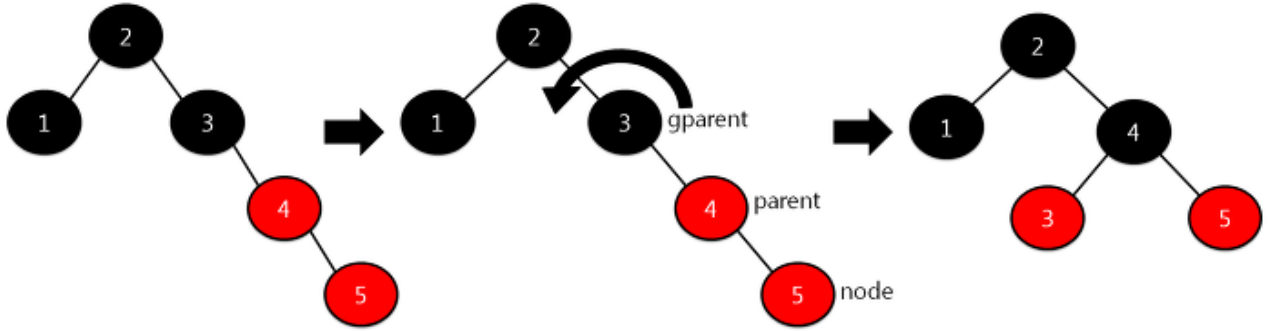
- classic red-black tree case by Rue 4 – red node 는 red node 를 자식으로 가질 수 없음
  - 1 ~ 3 insert




  - insert 4

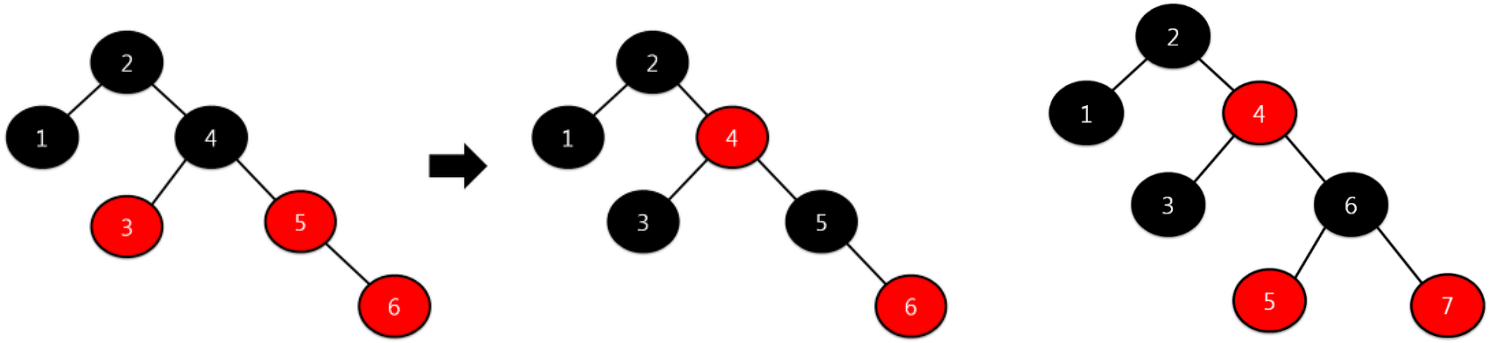



  - insert 5

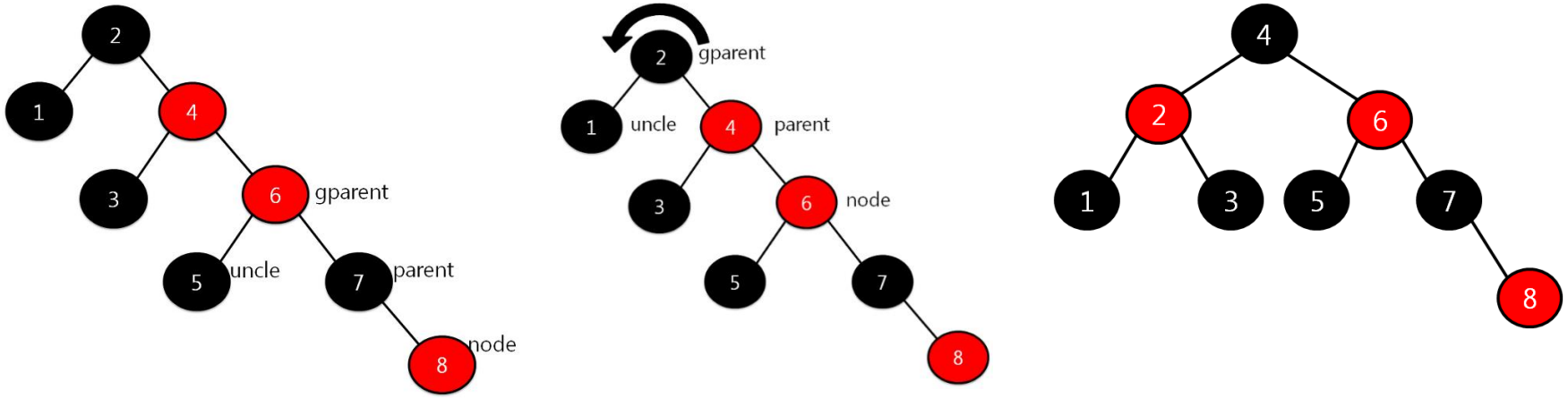


# Appendix – Red Black Tree

- classic red-black tree case by Rue 4 – red node 는 red node 를 자식으로 가질 수 없음
  - 6,7 insert




  - 8 insert

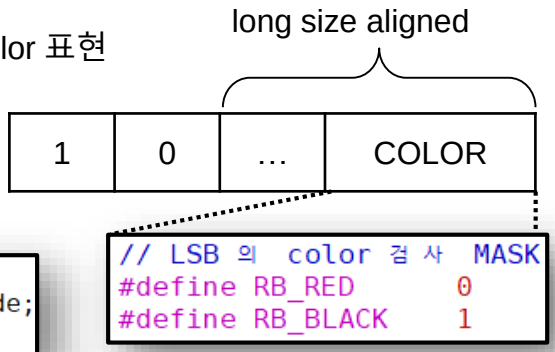


# Appendix – Red Black Tree

- classic red-black tree code
  - 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 비어있는지 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 code
  - 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 삽입할 때 불려지기 위한 함수로 __rb_parent_color 를
    // parent 로 설정한다는 것은 node 의 color 를 red 로 설정한다는
    // 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
    // 넘겨주어 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
  - 각 node 마다 추가적인 정보를 활용하여 추가, 검색, 삭제 등의 시간을 줄이기 위한 rbtree 의 확장
  - 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 를 사용한 mm 에서의 vma 관리
  - 각 mm 마다 관리하는 vma 는 augmented rbtree 를 사용하여 관리하며 augmented value 로 vma->rb_subtree_gap 사용
  - vma->rb_subtree_gap : rb_left, rb_right 각각의 subtree 에 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 vma 의 vm_end 와 현 재 vma 의 vm_start 사 이 에
    // 얼 만 큼 의 free 가 있 는 지 구 함
    if (vma->vm_rb.rb_left) {
        // left child 가 있 다 면 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;
        // 현 제 vma 값 과 비 교 하 여 최 대 값 구 함
    }
    if (vma->vm_rb.rb_right) {
        // 오 른 쪽 자 식 도 마 찬 가 지 작 업 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 mm 마다 관리하는 vma 는 augmented rbtree 를 사용하여 관리하며 augmented value 로 vma->rb_subtree_gap 사용
  - vma->rb_subtree_gap : rb_left, rb_right 각각의 subtree 에 존재하는 free virtual address space 크기 중 최대 값

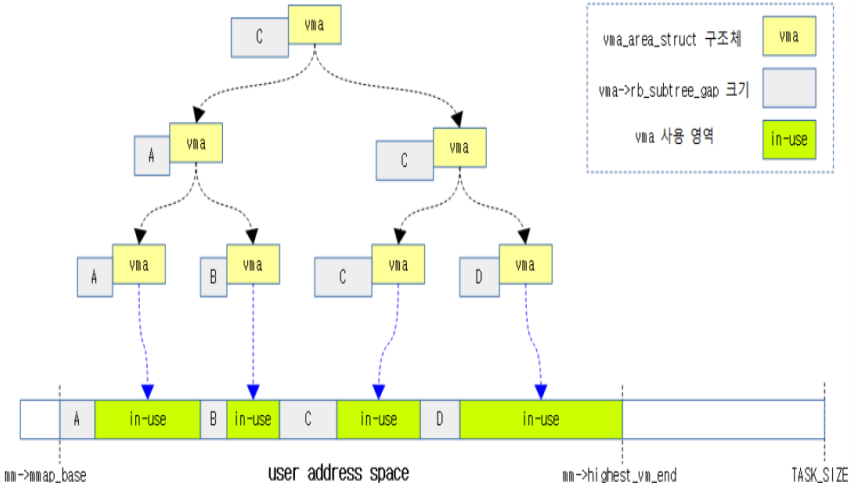
```
#define RB_DECLARE_CALLBACKS(rbstatic, rbname, rbstruct, rbfield,   \
                 rbtype, rbaugmented, rbcompute)        \
static inline void                                      \
rbname ## _propagate(struct rb_node *rb, struct rb_node *stop)      \
{                                                       \
    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 node 에 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);
        // vma 의 augmented 값을 계산하여 가져옴
        if (node->rbaugmented == augmented)
            break;
        // 가져와 계산한 값이 달라진다면
        node->rbaugmented = augmented;
        // 값 update 하고
        rb = rb_parent(&node->vm_rb);
        // parent node 로 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_old 의 augmented value 를 new 로 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);
    // old 의 값은 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 mm 마다 관리하는 vma 는 augmented rbtree 를 사용하여 관리하며 augmented value 로 vma->rb_subtree_gap 사용
  - vma->rb_subtree_gap : rb_left, rb_right 각각의 subtree 에 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 node 에 도달할 때까지 반복
        struct vm_area_struct *node = rb_entry(rb, struct vm_area_struct, vm_rb);
        unsigned long long augmented = vm_compute_subtree_gap(node);
        // vma 의 augmented 값을 계산하여 가져옴
        if (node->rbaugmented == augmented)
            break;
        // 가져와 계산한 값이 달라진다면
        node->rbaugmented = augmented;
        // 값 update 하고
        rb = rb_parent(&node->vm_rb);
        // parent node 로 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_old 의 augmented value 를 new 로 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);
    // old 의 값은 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  첫번째  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);
    // task_struct 별로  가지고  있는
    if (likely(vma))
        return vma;
    // cache 에  없으므로  이제  rb tree 에서  찾아야  함
    // 일단  mm 에서  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_node 에서  부터  순회하며  container_of 로  해당
        // 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 => addr 에  해당하는  vma 찾음
            rb_node = rb_node->rb_left;
            // case 1 => left child 로  이동
        } else
            rb_node = rb_node->rb_right;
            // case 3 => right child 로  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_struct 의  vmacache 의  sequence number 가  넘겨진
    // mm 가  가진  vm_area_struct 들인지  검사  아닐  경우및  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 찾은것
            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 첫번째 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 개임
    ...
}
```

```
struct vmacache {
    u32 seqnum;
    struct vm_area_struct *vmas[VMACACHE_SIZE];
};
```

```
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
    ...
}
```

```
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;
}
```

# 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 첫번째 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 개임
    ...
}
```

```
struct vmacache {
    u32 seqnum;
    struct vm_area_struct *vmas[VMACACHE_SIZE];
};
```

```
//  ...       --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
    // VMCACHE_SIZE 크기 (vm_area_struct 4개 )의
    // vmacache 에 해당하는 sequence number
    ...
}
```

```
        // case 3 => right child 로 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_struct 의 vmacache 의 sequence number 가 넘겨진
    // mm 가 가진 vm_area_struct 들인지 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 찾은것
            count_vm_vmacache_event(VMACACHE_FIND_HITS);
            return vma;
        }
    }

    return NULL;
}
```

# Operations on Regions

```
/* 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  첫번째  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);
    // task_struct 별로  가지고  있는
    if (likely(vma))
        return vma;
    // cache 에  없으므로  이제  rb tree 에서  찾아야  함
    // 일단  mm 에서  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_node 에서  부터  순회하며  container_of 로  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 => addr 에  해당하는  vma 찾음
            rb_node = rb_node->rb_left;
            // case 1 => left child 로  이동
        } else
            rb_node = rb_node->rb_right;
            // case 3 => right child 로  이동
    }
```

- 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 찾아지지 않을 경우 rbtree 의 node 를 순회하며 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

```
    // r1) exist
    // address 정확히 포함하는 vma 있음
    //                          *
    //    vm_end ~ vm_start---addr---vm_end ~ vm_start
    //      |        |   변수 vma      |        |
    //-------        ---------vm--------        ------
    //
    // r2) not exist
    // address 를 포함하는 vma 없고 address 가 다른 vma 사이에
    // 있음
    //                       *
    // vm_start---vm_end ~ addr ~ vm_start---vm_end ~ vm_start
    //   |          |               | 변수 vma |        |
    //   ----vm------               -----vm-----        ----
    //
    // r3) not exist (vma : null)
    // address 가 vma 들의 맨 끝에 있음 즉 현재 할당된 vma 들의
    // vma_end 들과 비교하여 제일 큰 위치
    //                       *
    // vm_start---vm_end ~ addr
    //   |          |
    //   -----vm-----
    //
    // r4) not exist (vma: null)
    // 처음 vma 없는 상태임
    //        *
    //      addr
    //
    // 검색 결과 vaddress 가 vma 들 사이에 있어
    //
    if (vma)
        vmacache_update(addr, vma);
    // 찾은 vma 를 task_struct 의 vmacache 에 추가

    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 찾아지지 않을 경우 rbtree 의 node 를 순회하며 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

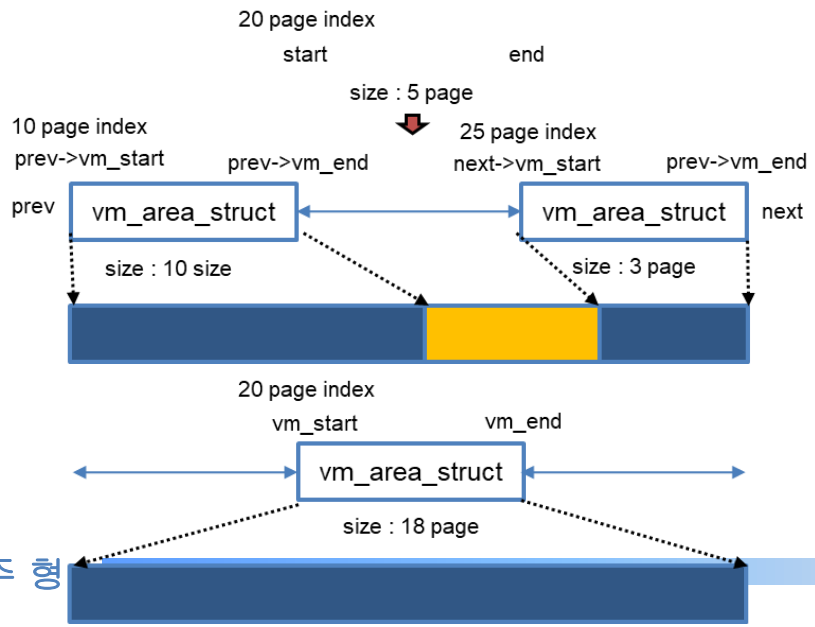
# Operations on Regions

- 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 region 들과 merge 될 수 있는지 확인하여 가능하다면 하나의 sequential region 으로 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_merage_after
        - filebacked -> 같은 file 인지
        - anonymous -> 같은 anon_vma 속하는지
    - 뒷 vm_area_struct 와 합쳐질 수 있는지 검사
    - ***__vma_adjust***

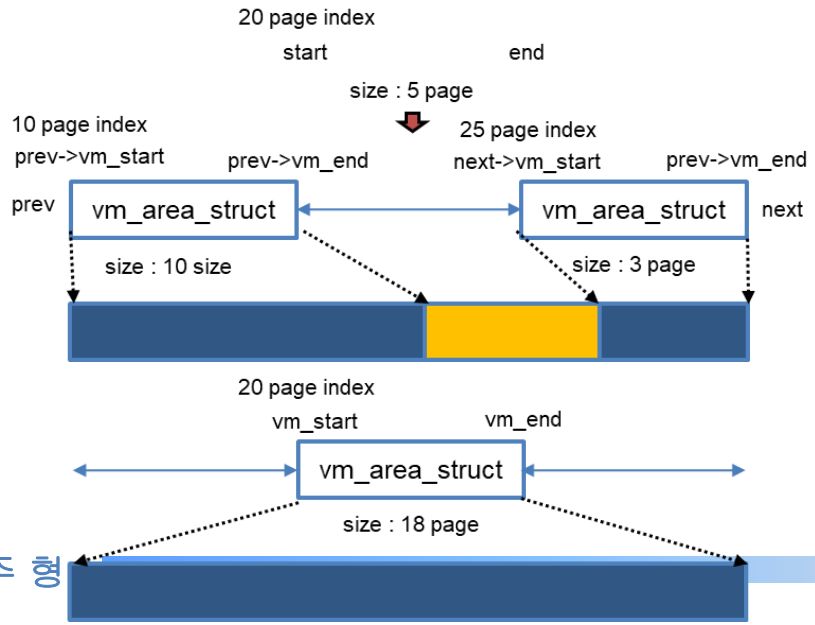



```
struct vm_area_struct *vma_merge(struct mm_struct *mm,
            struct vm_area_struct *prev, unsigned long addr,
            unsigned long end, unsigned long vm_flags,
            struct anon_vma *anon_vma, struct file *file,
            pgoff_t pgoff, struct mempolicy *policy,
            struct 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 page 라면  같은  file 인지 , 같은  속성의  vma 인지
        //     pg_off 기준으로  연속되는지  등등  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  vma 의  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 region 들과 merge 될 수 있는지 확인하여 가능하다면 하나의 sequential region 으로 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_merage_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,
```

```
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(anon_vma, vma->anon_vma, vma)) {
        // file backed 라면 같은 파일인지 , anon 이라면 같은 anon_vma 인지
        // 즉 같은 address space 인지
        pgoff_t vm_pglen;
        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 그전 prior vma 의 page 개수
        // 를 더한 값이 새로 추가될 vma 의 page offset 이 되어야 한다 .
        // (virtual address 가 연속적이어야 한다 .)
    }
    return 0;
}
```

```
                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_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

```
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_links 를 통해 vma 를 삽입할 mm 에서의 위치를 구함
    // rb_parent : rb_node 에서의 부모
    // rb_link : rb_parent 에서 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 Regio

- 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 첫번째 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 child 로 이동
        } else {
            rb_prev = __rb_parent; // case 6 => right child 로 이동
            // rb_prev 를 이동전 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_preb NULL 이면 처음 위치 (그림의 맨 왼쪽 처음자리 )
    *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);
}
```

```
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 node 로 올라가며 가장 큰 gap 크기 update
    else
        mm->highest_vm_end = vma->vm_end;
    // next 가 없다면 즉 vma 가 마지막이라면 mm 의 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 vma 의 vma->vm_rb 를 rb_parent 의 rb_link 위치 (left/right) 에
    // 연결하고 vma->vm_rb 의 parent pointer 에 rb_parent 설정
    vma->rb_subtree_gap = 0;
    vma_gap_update(vma);
    vma_rb_insert(vma, &mm->mm_rb);
    // balancing
}
```

```
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;
    // vma 에서 prev 연결
    if (prev) {
        next = prev->vm_next;
        prev->vm_next = vma;
        // prev 에서 vma 연결
    } else {
        // prev 가 없다면 맨 처음의 것이므로 mm->mmap 을 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;
    // vma 에서 next 로 연결 설정
    if (next)
        next->vm_prev = vma;
    // next 에서 vma 로 연결 설정
}
```

# 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);
}
```

```
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 node 로 올라가며 가장 큰 gap 크기 update
    else
        mm->highest_vm_end = vma->vm_end;
    // next 가 없다면 즉 vma 가 마지막이라면 mm 의 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.
```

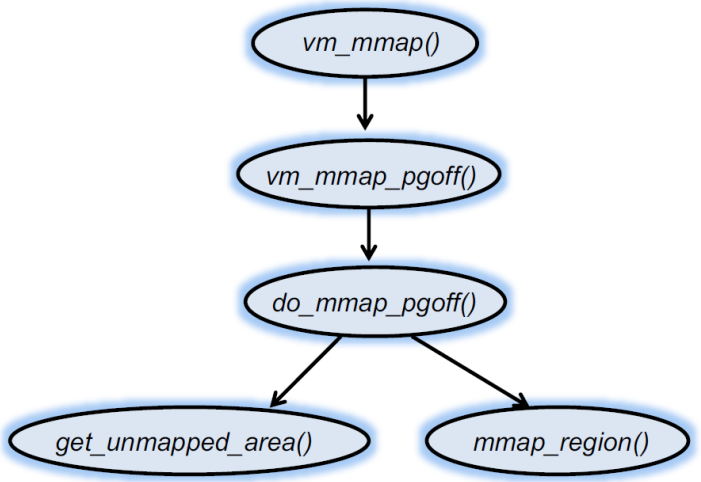
```
void __vma_link_list(struct mm_struct *mm, struct vm_area_struct *vma,
        struct vm_area_struct *prev, struct rb_node *rb_parent)
{
    struct vm_area_struct *next;
    vma->vm_prev = prev;
    // vma 에서 prev 연결
    if (prev) {
        next = prev->vm_next;
        prev->vm_next = vma;
        // prev 에서 vma 연결
    } else {
        // prev 가 없다면 맨 처음의 것이므로 mm->mmap 을 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;
    // vma 에서 next 로 연결 설정
    if (next)
        next->vm_prev = vma;
    // next 에서 vma 로 연결 설정
}
```

```
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 autmented rbtree 를 통해 추가 정보를 가진 gap 을 통해 free virtual address space 를 검색
    - ***unmap_area***
    - ***unmap_area_topdown***




```
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 거의뭐 3G 만큼 큰값인지 검사
    if (flags & MAP_FIXED)
        return addr;
    // addr 에 꼭 vma 생성해야 하는지 검사 그렇다면 그냥 반환
    // MAP_FIXED 는 mmap 함수에서 flag 로 주어 메모리 시작 번지
    // 지정 하려 할 때 사용
    if (addr) {
        // 주소가 명시되어 있다면 그 주소에 기존 vma 와겹치는지 확인해야함
        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 할당할 vma 의 끝주소가 TASK_SIZE 보다 작아야함
        // addr 위치가 minimum 보다 커야함
        // find_vma 를 통해 찾은 다음위치 vma 에 대해 vma 가 null 이어서 뒤에
        // 아무것도 없으므로 할당 가능한 상태이거나 , 뒤에 vma 가 있다면
        // vma->start 가 addr+len 보다 커야 함 즉 겹치면 안됨
    }
    // 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 을 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

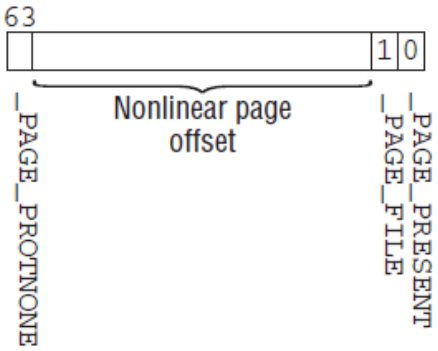



Figure 4-14: Representing nonlinear mappings in page table entries on IA-64 systems.

# 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
  - 책의 v2.6.39 에서는 object based reverse mapping (priority based tree) 가 사용되었지만 v4.11 확인 결과 interval rbtree 가 사용됨
    - interval tree vs prio tree
      - insert/erase : 25% faster
      - search : 2.4~3x faster

```
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;
#ifdef CONFIG_DEBUG_VM_RB
    unsigned long cached_vma_start, cached_vma_last;
#endif
};
```

```
struct anon_vma {
    struct anon_vma *root;      /* Root of this anon_vma tree */
    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;
    /*
     * 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;
    // 현재  aon_vma 를  가리키는  child anon_vma 의  개수
    struct anon_vma *parent;    /* Parent of this 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  rb tree
};
```

# Appendix – Interval Tree

- Interval Tree based Reverse Mapping

# Q & A