

VM Architecture – cont.

Jun-Chiao Wang, 22 Apr. 2002

NCTU CIS Operating System lab.

2002 Linux kernel trace seminar



Outline

- I The Slab Allocator
 - *kmem_cache_create*
 - *__kmem_cache_alloc*
- I Introduction to Red-Black tree
- I Process's address space
 - *do_mmap*
 - *do_munmap*

The Slab Allocator

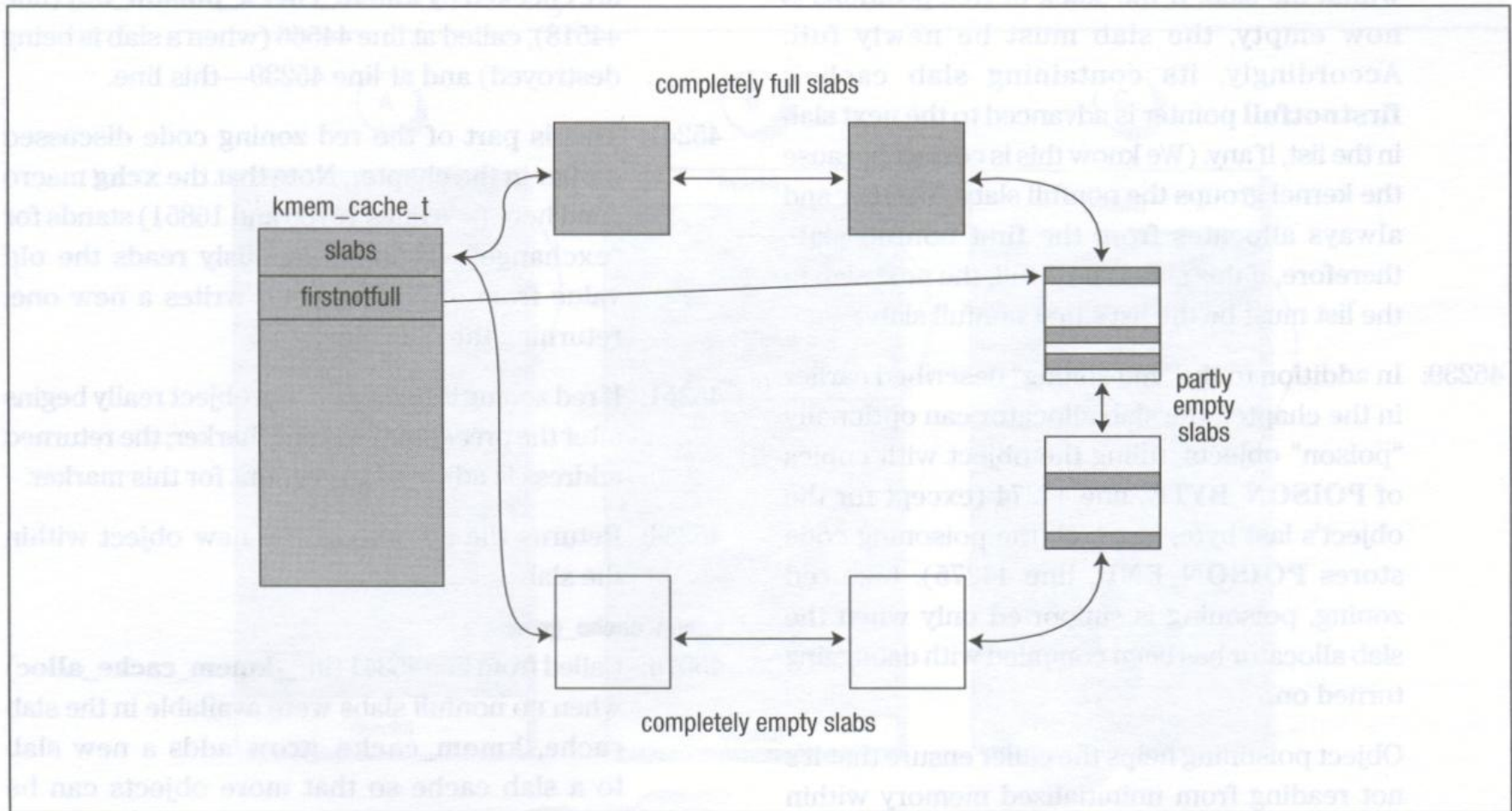


Figure 8.9 A slab cache and its list of slabs.

The Slab Allocator

cache data structure - kmem_cache_s (1)

```
struct kmem_cache_s {  
    /* 1) each alloc & free */  
    /* full, partial first, then free */  
    struct list_head slabs;  
    struct list_head *firstnotfull;  
    unsigned int objsize;  
    unsigned int flags;          /* constant flags */  
    unsigned int num;          /* # of objs per slab */  
  
    .....  
    /* 2) slab additions /removals */  
    /* order of pgs per slab (2^n) */  
    unsigned int gfporder;  
  
    /* force GFP flags, e.g. GFP_DMA */  
    unsigned int gfpflags;  
    .....  
};
```

The kernel manages *slab caches* with objects of type **kmem_cache_t** (which is just a **typedef** for **struct kmem_cache_s**)

The Slab Allocator

cache data structure - kmem_cache_s (2)

```
.....
kmem_cache_t *slabp_cache;
unsigned int growing;
.....

/* constructor func */
void (*ctor) (void *, kmem_cache_t *, unsigned long);

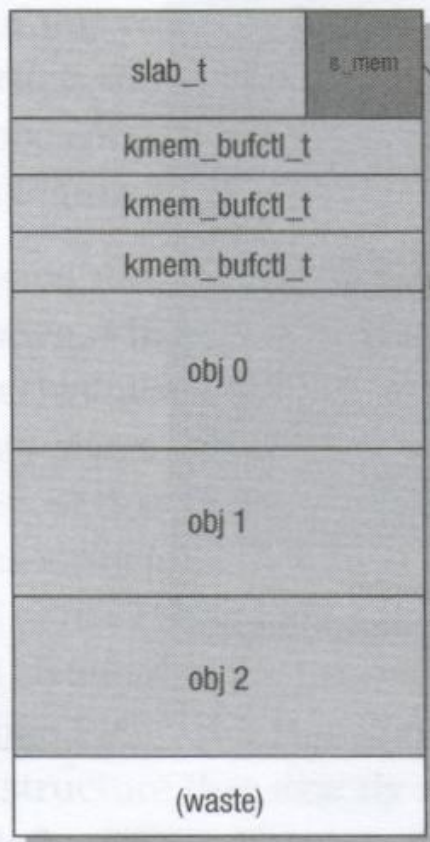
/* de-constructor func */
void (*dtor) (void *, kmem_cache_t *, unsigned long);
.....
/* 3) cache creation/removal */
char name[CACHE_NAMELEN];
struct list_head next;
.....
};
```

When a **new slab** is added to a slab **cache**, the slab cache's **ctor**, if non-NULL, is invoked for **each object pointer** within the slab.

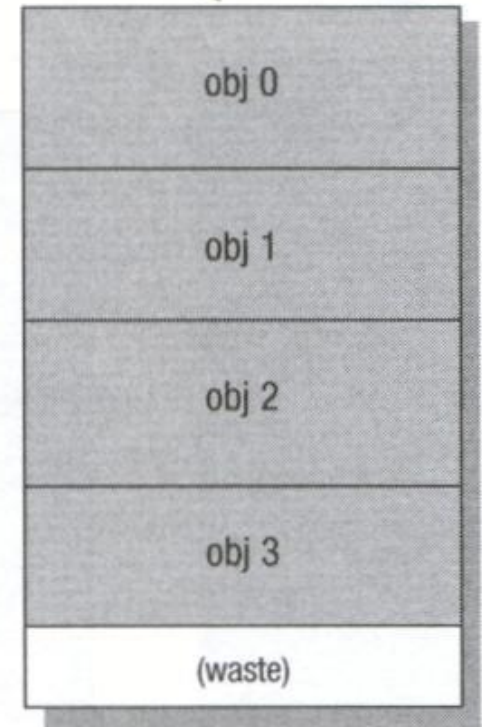
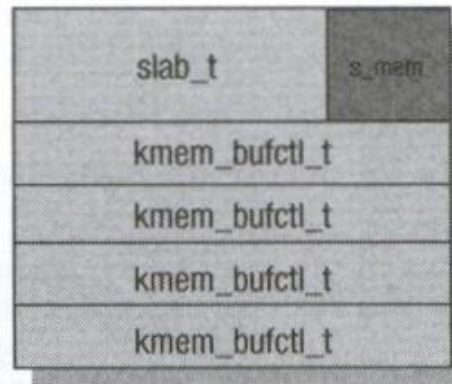
dtor is invoked for each object pointer just before the slab is released.

The Slab Allocator

two-kinds of slab



(On-slab slab)



(Off-slab slab)

The Slab Allocator

slab management information

```
| typedef struct slab_s {  
    struct list_head list;  
    .....  
    void *s_mem;  
    unsigned int inuse;  
    kmem_bufctl_t free;  
  
} slab_t;
```

The kernel uses a **slab_t** to track information about each slab in a slab

It keeps a stack of free positions within the slab, using a **kmem_bufctl_t** for each object on the slab

```
| typedef unsigned int kmem_bufctl_t;
```

kmem_bufctl_t is just the index of the next free position within the slab.

The Slab Allocator

General cache (1)

```
/* internal cache of cache description objs */
static kmem_cache_t cache_cache = {
    slabs:      LIST_HEAD_INIT(cache_cache.slabs),
    firstnotfull: &cache_cache.slabs,
    objsize:    sizeof(kmem_cache_t),
    flags:      SLAB_NO_REAP,
    spinlock:   SPIN_LOCK_UNLOCKED,
    colour_off: L1_CACHE_BYTES,
    name:       "kmem_cache",
};
```

I Initialized by `kmem_cache_init`

The Slab Allocator

General cache (2)

```
static cache_sizes_t cache_sizes[] = {
#ifdef PAGE_SIZE == 4096
    { 32,  NULL, NULL },
#endif
    { 64,  NULL, NULL },
    { 128, NULL, NULL },
    { 256, NULL, NULL },
    { 512, NULL, NULL },
    { 1024, NULL, NULL },
    { 2048, NULL, NULL },
    { 4096, NULL, NULL },
    { 8192, NULL, NULL },
    { 16384, NULL, NULL },
    { 32768, NULL, NULL },
    { 65536, NULL, NULL },
    { 131072, NULL, NULL },
    { 0,  NULL, NULL }
};
```

```
/* Size description struct for general
caches. */
typedef struct cache_sizes {
    size_t cs_size;
    kmem_cache_t *cs_cache;
    kmem_cache_t *cs_dmacache;
} cache_sizes_t;
```

- I Initialized by `kmem_cache_sizes_init()`

The Slab Allocator

kmem_cache_create (1)

```
kmem_cache_t *kmem_cache_create(const char *name,
    size_t size, size_t offset, unsigned long flags,
    void (*ctor) (void *, kmem_cache_t *, unsigned long),
    void (*dtor) (void *, kmem_cache_t *, unsigned long))
{
    const char *func_nm = KERN_ERR "kmem_create: ";
    size_t left_over, align, slab_size;
    kmem_cache_t *cachep = NULL;

    /* Sanity checks... these are all serious usage bugs.
     */
    .....

    /* Get cache's description obj. */
    cachep = (kmem_cache_t *)
        kmem_cache_alloc(&cache_cache, SLAB_KERNEL);
    if (!cachep)
        goto opps;
    memset(cachep, 0, sizeof(kmem_cache_t));
```

kmem_cache_create
creates a new slab cache.

The Slab Allocator

kmem_cache_create (2)

```
.....

align = BYTES_PER_WORD;
if (flags & SLAB_HWCACHE_ALIGN)
    align = L1_CACHE_BYTES;

/* Determine if the slab management is 'on' or 'off'
 * slab. */
if (size >= (PAGE_SIZE >> 3))
    /* Size is large, assume best to place the slab
     * management obj off-slab (should allow better
     * packing of objs). */
    flags |= CFLGS_OFF_SLAB;

if (flags & SLAB_HWCACHE_ALIGN) {
    while (size < align / 2)
        align /= 2;
    size = (size + align - 1) & ~(align - 1);
}
```

The Slab Allocator

kmem_cache_create

```
do {
    unsigned int break_flag = 0;
cal_wastage:
    kmem_cache_estimate(cachep->gfporder,
                        &left_over, &cachep->num);
    if (break_flag)
        break;
    if (cachep->gfporder >= MAX_GFP_ORDER)
        break;
    if (!cachep->num)
        goto next;
    if (flags & CFLGS_OFF_SLAB
        && cachep->num > offslab_limit) {
        cachep->gfporder - -;
        break_flag++;
        goto cal_wastage;
    }
}
```

```
static void kmem_cache_estimate(unsigned long gfporder,
                                size_t size, int flags, size_t * left_over,
                                unsigned int *num)
{
    int i;
    size_t wastage = PAGE_SIZE << gfporder;
    size_t extra = 0;
    size_t base = 0;

    if (!(flags & CFLGS_OFF_SLAB)) {
        base = sizeof(slab_t);
        extra = sizeof(kmem_bufctl_t);
    }
    i = 0;
    while (i * size + L1_CACHE_ALIGN(base + i * extra) <=
           wastage)
        i++;
    if (i > 0)
        i--;

    if (i > SLAB_LIMIT)
        i = SLAB_LIMIT;

    *num = i;
    wastage -= i * size;
    wastage -= L1_CACHE_ALIGN(base + i * extra);
    *left_over = wastage;
}
```

The Slab Allocator

kmem_cache_create (4)

```
if (cachep->gfporder >= slab_break_gfp_order)
    break;

if ((left_over * 8) <=
    (PAGE_SIZE << cachep->gfporder))
    break; /* Acceptable internal fragmentation. */
    .....
next:
    cachep->gfporder++;
} while (1);

if (!cachep->num) {
    printk("kmem_cache_create: couldn't create cache "
        "%s.\n", name);
    kmem_cache_free(&cache_cache, cachep);
    cachep = NULL;
    goto opps;
}
slab_size = L1_CACHE_ALIGN(cachep->num *
    sizeof(kmem_bufctl_t) + sizeof(slab_t));
```

The Slab Allocator

kmem_cache_create (5)

```
if (flags & CFLGS_OFF_SLAB && left_over >= slab_size) {
    flags &= ~CFLGS_OFF_SLAB;
    left_over -= slab_size;
}
```

.....

```
/* init remaining fields */
```

.....

```
cachep->flags = flags;
cachep->gfpflags = 0;
if (flags & SLAB_CACHE_DMA)
    cachep->gfpflags |= GFP_DMA;
```

.....

```
cachep->objsize = size;
INIT_LIST_HEAD(&cachep->slabs);
cachep->firstnotfull = &cachep->slabs;
```

The Slab Allocator

kmem_cache_create (6)

```
if (flags & CFLGS_OFF_SLAB)
    cachep->slabp_cache =
        kmem_find_general_cachep(slabp_cache, flags);
cachep->ctor = ctor;
cachep->dtor = dtor;
/* Copy name over so we don't have problems with
 * unloaded modules */
strcpy(cachep->name, name);
.....

list_add(&cachep->next, &cache_chain);
.....

oops:
return cachep;
}
```

```
kmem_cache_t *kmem_find_general_cachep(size_t size,
int gfpflags)
{
    cache_sizes_t *csizp = cache_sizes;

    for (; csizp->cs_size; csizp++) {
        if (size > csizp->cs_size)
            continue;
        break;
    }
    return (gfpflags & GFP_DMA)
        ? csizp->cs_dmacachep
        : csizp->cs_cachep;
}
```

The Slab Allocator

kmem_cache_alloc & *kmalloc*

```
void *kmem_cache_alloc(kmem_cache_t * cachep, int flags)
{
    return __kmem_cache_alloc(cachep, flags);
}

void *kmalloc(size_t size, int flags)
{
    cache_sizes_t *csizep = cache_sizes;

    for (; csizep->cs_size; csizep++) {
        if (size > csizep->cs_size)
            continue;
        return __kmem_cache_alloc(flags & GFP_DMA
                                   ? csizep->cs_dmacachep
                                   : csizep->cs_cachep, flags);
    }
    BUG();           // too big size
    return NULL;
}
```

The Slab Allocator

__kmem_cache_alloc(1)

```
static inline void *__kmem_cache_alloc
{
    .....
    void *objp;
    kmem_cache_alloc_head(cachep,
try_again:
    .....
#ifdef CONFIG_SMP
    .....
#else
    objp = kmem_cache_alloc_c
#endif
    .....
    return objp;
alloc_new_slab:
    .....
    if (kmem_cache_grow(cachep, flags))
        goto try_again;
    return NULL;
}
```

```
#define kmem_cache_alloc_one(cachep)
({
    slab_t *slabp;

    /* Get slab alloc is to come from. */
    {
        struct list_head* p = cachep->firstnotfull;
        if (p == &cachep->slabs)
            goto alloc_new_slab;
        slabp = list_entry(p, slab_t, list);
    }
    kmem_cache_alloc_one_tail(cachep, slabp);
})
```

The Slab Allocator

`__kmem_cache_alloc (2)-`

`kmem_cache_alloc_one_tail`

```
static inline void * kmem_cache_alloc_one_tail(kmem_cache_t * cachep, slab_t * slabp)
{
    void *objp;
        .....

    /* get obj pointer */
    slabp->inuse++;
    objp = slabp->s_mem + slabp->free * cachep->objsize;
    slabp->free = slab_bufctl(slabp)[slabp->free];

    if (slabp->free == BUFCTL_END)
        /* slab now full: move to next slab for next alloc
        cachep->firstnotfull = slabp->list.next;
        .....
    return objp;
}
```

```
#define slab_bufctl(slabp) \
    ((kmem_bufctl_t) (((slab_t*)slabp)+1))
```

The Slab Allocator

`__kmem_cache_alloc (3)-`

`kmem_cache_grow(1)`

```
static int kmem_cache_grow(kmem_cache_t * cachep, int flags)
```

```
{
```

```
    slab_t *slabp;
```

```
    .....
```

```
    void *objp;
```

```
    .....
```

```
    cachep->growing++;
```

```
    /* Get mem for the objs. */
```

```
    if (!(objp = kmem_getpages(cachep,  
                                goto failed;
```

```
    /* Get slab management. */
```

```
    if (!(slabp = kmem_cache_slabmgmt(cachep, objp, offset,  
                                       local_flags)))
```

```
        goto opps1;
```

```
static inline void *kmem_getpages(kmem_cache_t *  
                                   cachep, unsigned long flags)
```

```
{
```

```
    void *addr;
```

```
    flags |= cachep->gfpflags;
```

```
    addr = (void *) __get_free_pages(flags, cachep->  
                                       >gfporder);
```

```
    return addr;
```

```
}
```

The Slab Allocator

__kmem_cache_alloc (4)-

kmem_cache_grow(2)

```
kmem_cache_init_objs(cachep, slabp, ctor_flags);
```

```
cachep->growing--;
```

```
/* Make slab active. */
```

```
list_add_tail(&slabp->list, &cachep->slabs);
```

```
if (cachep->firstnotfull == &cachep->slabs)
```

```
    cachep->firstnotfull = &slabp->list;
```

```
    .....
```

```
    return 1;
```

```
oops1:
```

```
    kmem_freepages(cachep, objp);
```

```
failed:
```

```
    .....
```

```
    cachep->growing--;
```

```
    .....
```

```
    return 0;
```

```
}
```

The Slab Allocator

`__kmem_cache_alloc (5)-`

`kmem_cache_init_objs`

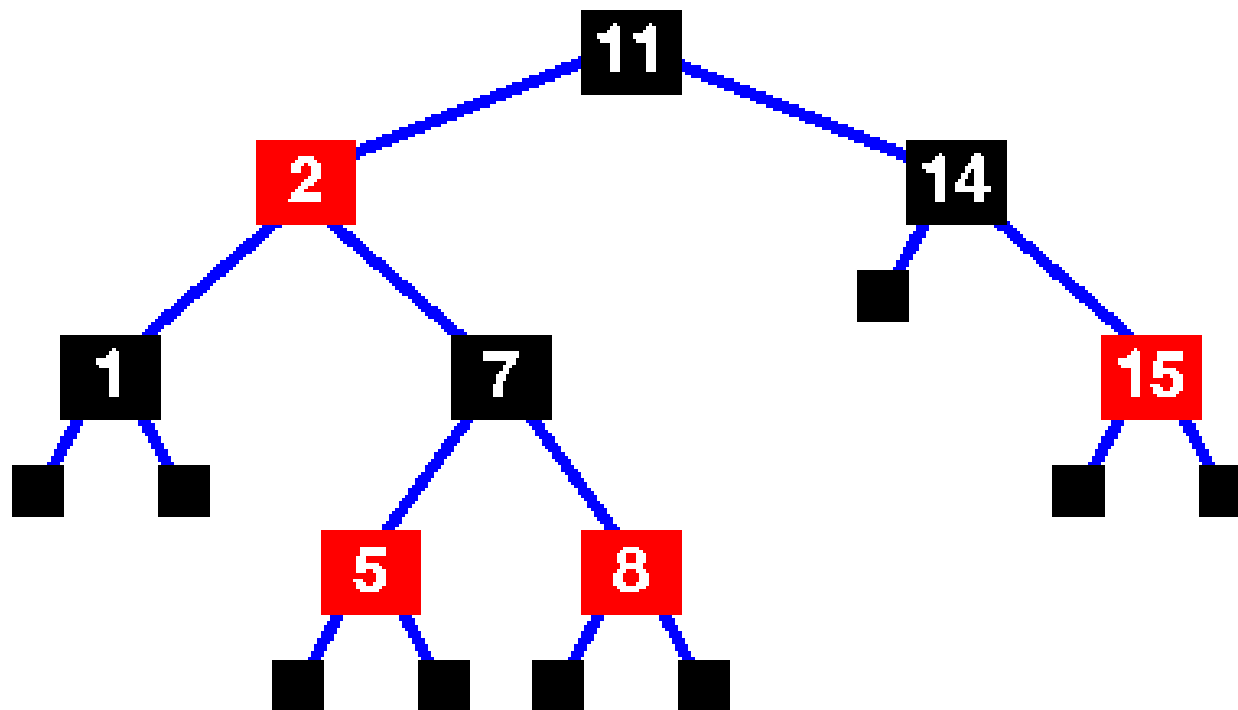
```
static inline void kmem_cache_init_objs( kmem_cache_t * cachep, slab_t * slabp, unsigned long
    ctor_flags)
{
    int i;

    for (i = 0; i < cachep->num; i++) {
        void *objp = slabp->s_mem + cachep->objsize * i;
        .....
        if (cachep->ctor)
            cachep->ctor(objp, cachep, ctor_flags);
        .....
        slab_bufctl(slabp)[i] = i + 1;
    }
    slab_bufctl(slabp)[i - 1] = BUFCTL_END;
    slabp->free = 0;
}
```

Intro. To RB-tree

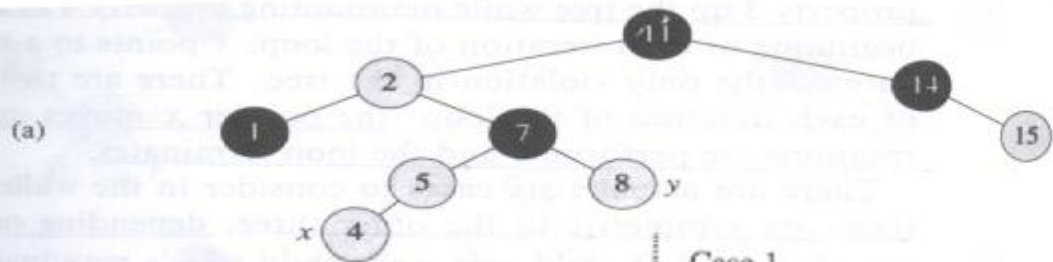
- | Red-black trees are one of many search-tree schemes that are “*balanced*” in order to guarantee that basic dynamic-set operations take $O(\log n)$ time in the worst case.
- | A red-black tree is a binary search tree with one extra bit of storage per node: its *color* (Red or Black)
- | A binary search tree is a red-black tree if it satisfies the following red-black properties:
 - Every node is either red or black
 - Every leaf (NIL) is black
 - If a node is red , then both its children are black
 - Every simple path from a node to a descendant leaf contains the same number of black nodes

Intro. To RB-tree

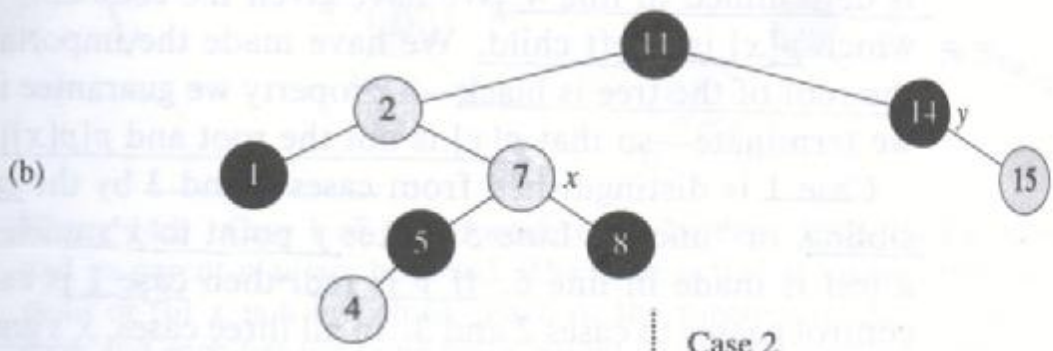


RB-INSERT(T, x)

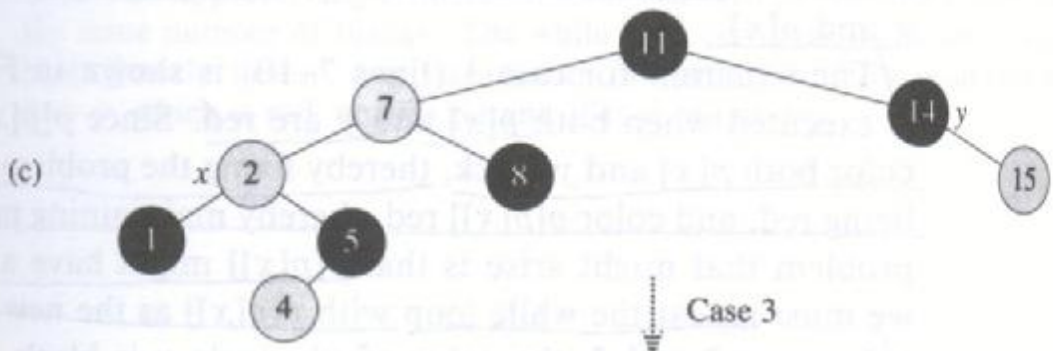
```
1  TREE-INSERT( $T, x$ )
2   $color[x] \leftarrow RED$ 
3  while  $x \neq root[T]$  and  $color[p[x]] = RED$ 
4      do if  $p[x] = left[p[p[x]]]$ 
5          then  $y \leftarrow right[p[p[x]]]$ 
6              if  $color[y] = RED$ 
7                  then  $color[p[x]] \leftarrow BLACK$            ▷ Case 1
8                      $color[y] \leftarrow BLACK$              ▷ Case 1
9                      $color[p[p[x]]] \leftarrow RED$          ▷ Case 1
10                     $x \leftarrow p[p[x]]$                    ▷ Case 1
11                else if  $x = right[p[x]]$ 
12                    then  $x \leftarrow p[x]$                  ▷ Case 2
13                    LEFT-ROTATE( $T, x$ )                     ▷ Case 2
14                     $color[p[x]] \leftarrow BLACK$          ▷ Case 3
15                     $color[p[p[x]]] \leftarrow RED$          ▷ Case 3
16                    RIGHT-ROTATE( $T, p[p[x]]$ )             ▷ Case 3
17                else (same as then clause
18                    with “right” and “left” exchanged)
18   $color[root[T]] \leftarrow BLACK$ 
```



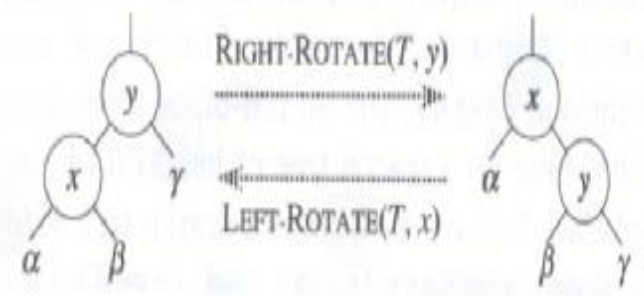
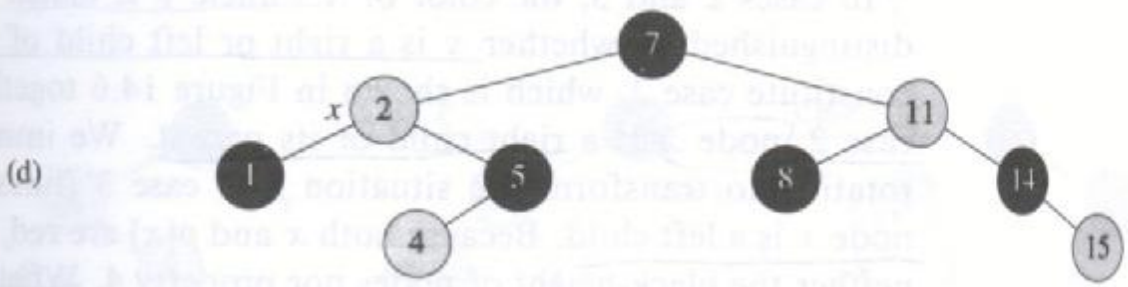
Case 1



Case 2



Case 3



Intro. To RB-tree

- *Characteristics*

- I A red-black tree with n internal nodes has height at most $2\log(n+1)$
- I RB-Insert
 - The while loop only repeats if case 1 is executed, and then the pointer x moves up the tree.
 - It never performs more than two rotations, since the while loop terminates if case 2 or case 3 is executed.
- I RB-Delete
 - performing a constant number of the while loop and at most three rotations.

Process's address space

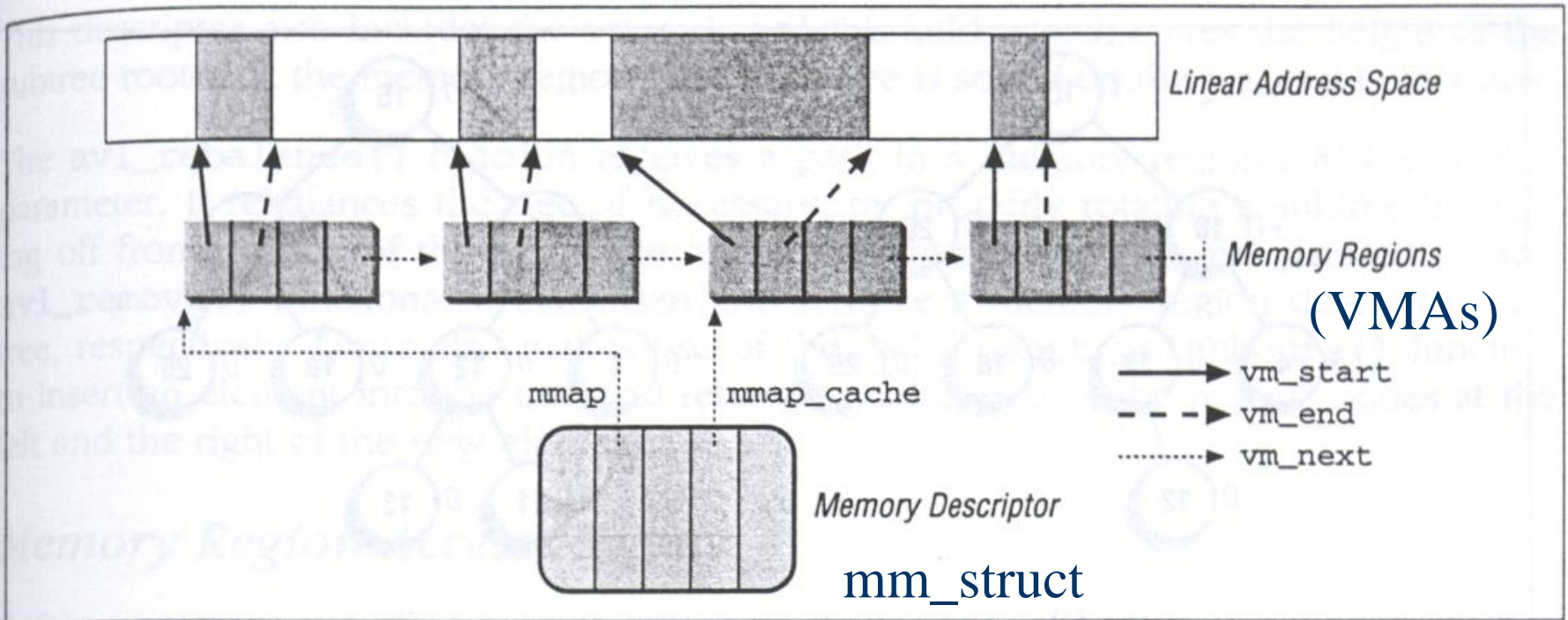


Figure 7-2. Descriptors related to the address space of a process

Data structure

```
struct vm_area_struct {
```

```
    struct mm_struct * vm_mm;  
    unsigned long vm_start;  
    unsigned long vm_end;
```

```
    /* linked list of VM areas per task  
    struct vm_area_struct * vm_next;  
    unsigned long vm_flags;  
    rb_node_t vm_rb;
```

```
    .....
```

```
};
```

The kernel tracks the memory areas used by a process with zero or more **struct vm_area_structs**, commonly abbreviated as VMAs

```
/include/linux/rbtree.h
```

```
typedef struct rb\_node\_s
```

```
{
```

```
    struct rb\_node\_s * rb_parent;  
    int rb_color;
```

```
#define RB\_RED 0
```

```
#define RB\_BLACK 1
```

```
    struct rb\_node\_s * rb_right;  
    struct rb\_node\_s * rb_left;
```

```
} rb\_node\_t;
```

Data structure

```
struct mm_struct {
    struct vm_area_struct * mmap;           /* list of VMAs */
    rb_root_t mm_rb;                       /* last find_vma result */
    struct vm_area_struct * mmap_cache;
    pgd_t * pgd;
    atomic_t mm_users;                     /* How many users with user space? */
    atomic_t mm_count;                    /* How many references to "struct
                                           mm_struct" (users count as 1) */

    int map_count;                         /* number of VMAs */
    .....
    unsigned long start_code, end_code, start_data, end_data;
    unsigned long start_brk, brk, start_stack;
    .....
};
```

find_vma (1)

```
struct vm_area_struct * find_vma(struct mm_struct * mm, unsigned long addr)
{
    struct vm_area_struct *vma = NULL;

    if (mm) {
        /* Check the cache first. */
        /* (Cache hit rate is typically around 35%.) */
        vma = mm->mmap_cache;
        if (!(vma && vma->vm_end > addr && vma->vm_start <= addr)) {
            rb_node_t * rb_node;

            rb_node = mm->mm_rb.rb_node;
            vma = NULL;
        }
    }
}
```

| *find_vma*

- Look up the first VMA which satisfies **addr** < **vm_end**, return **NULL** if none.

find_vma (2)

```
while (rb_node) {
    struct vm_area_struct * vma_tmp;

    vma_tmp = rb_entry(rb_node, struct vm_area_struct, vm_rb);

    if (vma_tmp->vm_end > addr) {
        vma = vma_tmp;
        if (vma_tmp->vm_start <= addr)
            break;
        rb_node = rb_node->rb_left;
    } else
        rb_node = rb_node->rb_right;
}
if (vma)
    mm->mmap_cache = vma;
}
return vma;
}
```

do_mmap

```
static inline unsigned long do_mmap(struct file *file,
    unsigned long addr, unsigned long len,
    unsigned long prot, unsigned long flag,
    unsigned long offset)
{
    unsigned long ret = -EINVAL;
    if ((offset + PAGE_ALIGN(len)) < 0)
        goto out;
    if (!(offset & ~PAGE_MASK))
        ret = do_mmap_pgoff(file, addr, len, prot, flag,
            offset >> PAGE_SHIFT);
out:
    return ret;
}
```

/ to align the pointer to the (next) page boundary */*

```
#define PAGE_ALIGN(addr)
(((addr)+PAGE_SIZE-1)&PAGE_MASK)
```

do_mmap_pgoff (1)

unsigned long **do_mmap_pgoff**(struct file *file, unsigned long addr, unsigned long len, unsigned long prot, unsigned long flags, unsigned long pgoff)

```
{
    struct mm_struct *mm = current->mm;
    struct vm_area_struct *vma;
    .....
    rb_node_t ** rb_link, * rb_parent;
    .....
    if ((len = PAGE_ALIGN(len)) == 0)
        return addr;
    if (len > TASK_SIZE || addr > TASK_SIZE - len)
        return -EINVAL;
    /* offset overflow? */
    if ((pgoff + (len >> PAGE_SHIFT)) < pgoff)
        return -EINVAL;
    /* Too many mappings? */
    if (mm->map_count > MAX_MAP_COUNT)
        return -ENOMEM;
```

do_mmap

```
/* Obtain the address of the first unmapped page
 * that it represents.
 */
addr = get_unmapped_area(file, addr, len, pgoff, flags);
if (addr & ~PAGE_MASK)
    return addr;
.....
vma = kmem_cache_alloc(vm_area_caches[0], GFP_KERNEL);
if (!vma)
    return -ENOMEM;
vma->vm_mm = mm;
vma->vm_start = addr;
vma->vm_end = addr + len;
vma->vm_flags = VM_READ | VM_WRITE | VM_EXEC;
.....
}

/mm/mmap.c
unsigned long get_unmapped_area (.....)
{
    .....
    if (file && file->f_op && file->f_op->get_unmapped_area)
        return file->f_op->get_unmapped_area(file, addr, len, pgoff,
flags);
    return arch_get_unmapped_area(file, addr, len, pgoff, flags);
}
static inline unsigned long arch_get_unmapped_area (.....)
{
    struct vm_area_struct *vma;
    .....
    if (len > TASK_SIZE)
        return -ENOMEM;
    .....
    for (vma = find_vma(current->mm, addr); ; vma = vma-
>vm_next) {
        if (TASK_SIZE - len < addr)
            return -ENOMEM;
        if (!vma || addr + len <= vma->vm_start)
            return addr;
        addr = vma->vm_end;
    }
}
```

do_mmap_pgoff (3)

```
.....  
addr = vma->vm_start;  
vma_link(mm, vma, prev, rb_parent);  
.....  
free_vma:  
kmem_cache_free(vm_area_cache, vma);  
return error;  
}
```

“vma_link” calls “__vma_link”

```
static void __vma_link(struct mm_struct * mm, struct  
vm_area_struct * vma, struct vm_area_struct * prev,  
rb_node_t ** rb_link, rb_node_t * rb_parent)  
{
```

```
    __vma_link_list(mm, vma, prev, rb_parent);  
    __vma_link_rb(mm, vma, rb_link, rb_parent);  
    __vma_link_file(vma);  
}
```

```
static inline void __vma_link_rb(struct mm_struct * mm,  
struct vm_area_struct * vma,  
rb_node_t ** rb_link, rb_node_t * rb_parent)
```

```
{  
    rb_link_node(&vma->vm_rb, rb_parent, rb_link);  
    rb_insert_color(&vma->vm_rb, &mm->mm_rb);  
}
```

do_munmap (1)

```
int do_munmap(struct mm_struct *mm, unsigned long a
{
    struct vm_area_struct *mpnt, *prev, **npp, *free *

    if ((addr & ~PAGE_MASK) || addr > TASK_SIZE || len >
        return -EINVAL;

    if ((len = PAGE_ALIGN(len)) == 0)
        return -EINVAL;

    mpnt = find_vma_prev(mm, addr, &prev);
    if (!mpnt)
        return 0;
    /* we have addr < mpnt->vm_end */

    if (mpnt->vm_start >= addr+len)
        return 0;
```

If the address is not page-aligned or the region lies outside the process's memory space, it's clearly invalid.

Freeing 0 bytes would be an error. Otherwise, **len** is rounded up so that entire pages will be freed.

Finds the VMA that includes the given address.

do_munmap (2)

```
if ((mpnt->vm_start < addr && mpnt->vm_end > addr+len)
    && mm->map_count >= MAX_MAP_COUNT)
    return -ENOMEM;
```

We may need one additional vma
to fix up the mappings ...

```
extra = kmem_cache_alloc(vm_area_cachep, SLAB_KERNEL);
```

```
if (!extra)
    return -ENOMEM;
```

```
npp = (prev ? &prev->vm_next : &mm->mmap);
free = NULL;
spin_lock(&mm->page_table_lock);
for ( ; mpnt && mpnt->vm_start < addr+len; mpnt = *npp) {
    *npp = mpnt->vm_next;
    mpnt->vm_next = free;
    free = mpnt;
    rb_erase(&mpnt->vm_rb, &mm->mm_rb);
}
```

do_munmap (3)

```
mm->mmap_cache = NULL; /* Kill the cache */
spin_unlock(&mm->page_table_lock);

while ((mpnt = free) != NULL) {
    unsigned long st, end, size;
    .....
    free = free->vm_next;

    st = addr < mpnt->vm_start ? mpnt->vm_start : addr;
    end = addr+len;
    end = end > mpnt->vm_end ? mpnt->vm_end : end;
    size = end - st;
    .....
    mm->map_count--;
    zap_page_range(mm, st, size);
    extra = unmap_fixup(mm, mpnt, st, size, extra);
    .....
}
.....
```

These three lines could be written as follows:

```
st = max (mpnt ->vm_start ,
          addr );
end = min (mpnt->vm_end ,
          addr+ len );
```

Hence, **st** is the start of the region that **do_munmap** *actually* starts freeing up and **end** is the regions' end.

Updates the MMU data structures corresponding to the current subregion that is being freed up within this VMA

do_munmap (4)

```
if (extra)
    kmem_cache_free(vm_area_cachep, extra);

free_pgtables(mm, prev, addr, addr+len);

return 0;
}
```

If **extra** is NULL , the descriptor has been used ; otherwise , **do_munmap ()** involves **kmem_cache_free** to release it.

do_munmap (5) - *unmap_fixup* (1)

```
static struct vm_area_struct * unmap_fixup(struct mm_struct *mm,
      struct vm_area_struct *area, unsigned long addr, size_t len,
      struct vm_area_struct *extra)
{
    struct vm_area_struct *mpnt;
    unsigned long end = addr + len;
    .....
    /* Unmapping the whole area. Case 1 */
    if (addr == area->vm_start && end == area->vm_end) {
        .....
        kmem_cache_free(vm_area_cachep, area);
        return extra;
    }
}
```

do_munmap (6) - unmap_fixup (2)

```
if (end == area->vm_end) {                                /*Case2*/
    .....
    area->vm_end = addr;
    .....
} else if (addr == area->vm_start) {                       /*Case3*/
    .....
    area->vm_start = end;
    .....
} else {                                                  /*Case4*/
    mpnt = extra;
    extra = NULL;
    mpnt->vm_mm = area->vm_mm;
    mpnt->vm_start = end;
    mpnt->vm_end = area->vm_end;
    mpnt->vm_page_prot = area->vm_page_prot;
    mpnt->vm_flags = area->vm_flags;
    area->vm_end = addr; /* Truncate area */
    .....
}
```

do_munmap (7) - unmap_fixup (3)

```
.....
    __insert_vm_struct(mm, mpnt);
}
__insert_vm_struct(mm, area);          /* for case 2,3,4 update */
.....
return extra;
}
```

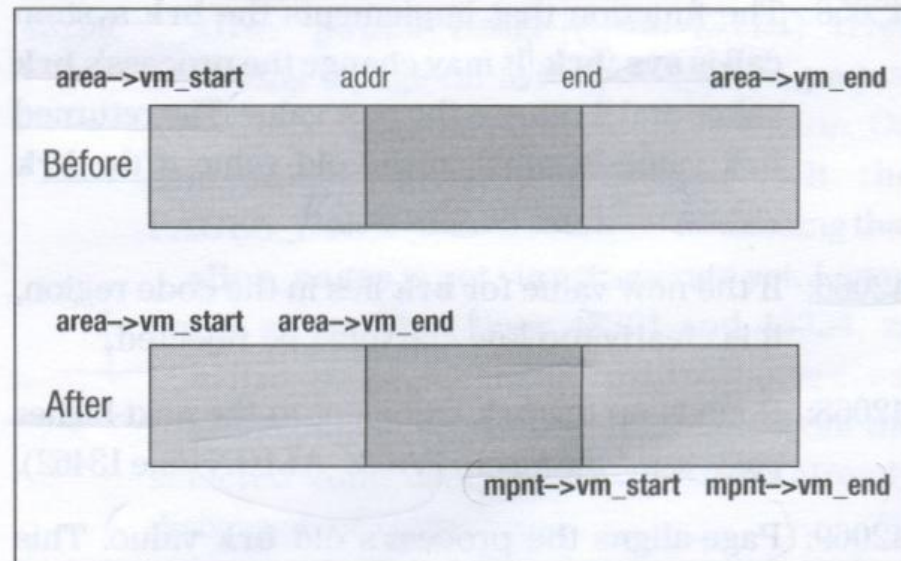


Figure 8.4 Splitting the VMA. (Case4)

References

- | **Linux Core Kernel Commentary second edition (CORIOLIS)**
- | **Understanding the LINUX KERNEL (O'reilly)**
- | **Cross-Referencing Linux**
 - <http://lxr.linux.no/>
- | **Introduction to Algorithms (MIT Press)**