

VM Architecture

Jun-Chiao Wang, 15 Apr. 2002

NCTU CIS Operating System lab.

2002 Linux kernel trace seminar



Outline

- I Introduction
 - What is VM?
 - Segmentation & Paging in Linux
- I Memory Management
 - Page Frame Management
 - The Buddy System
 - The Slab Allocator
 - vmalloc and vfree
- I Process's address space
 - Three data structure
 - Operating on VMAs

What is VM (Virtual Memory) ?

- | **VM** is the technique of seamlessly blending access to RAM and disk, primary and secondary storage.
- | **VM** also prefer to the practice of lying to processes about the addresses at which they reside.
- | **VM** acts as a logical layer between the application memory requests and the hardware Memory Management Unit (MMU) .

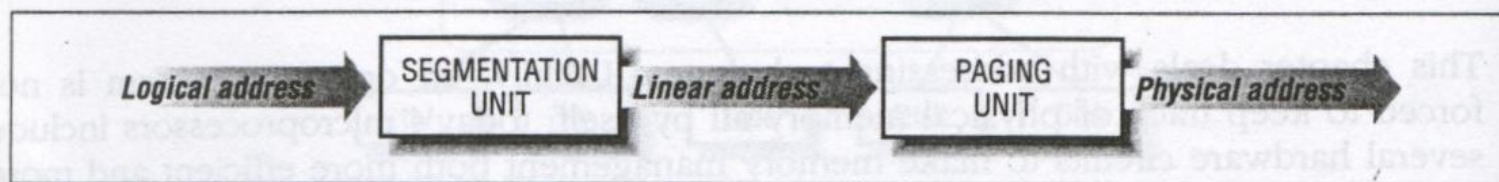
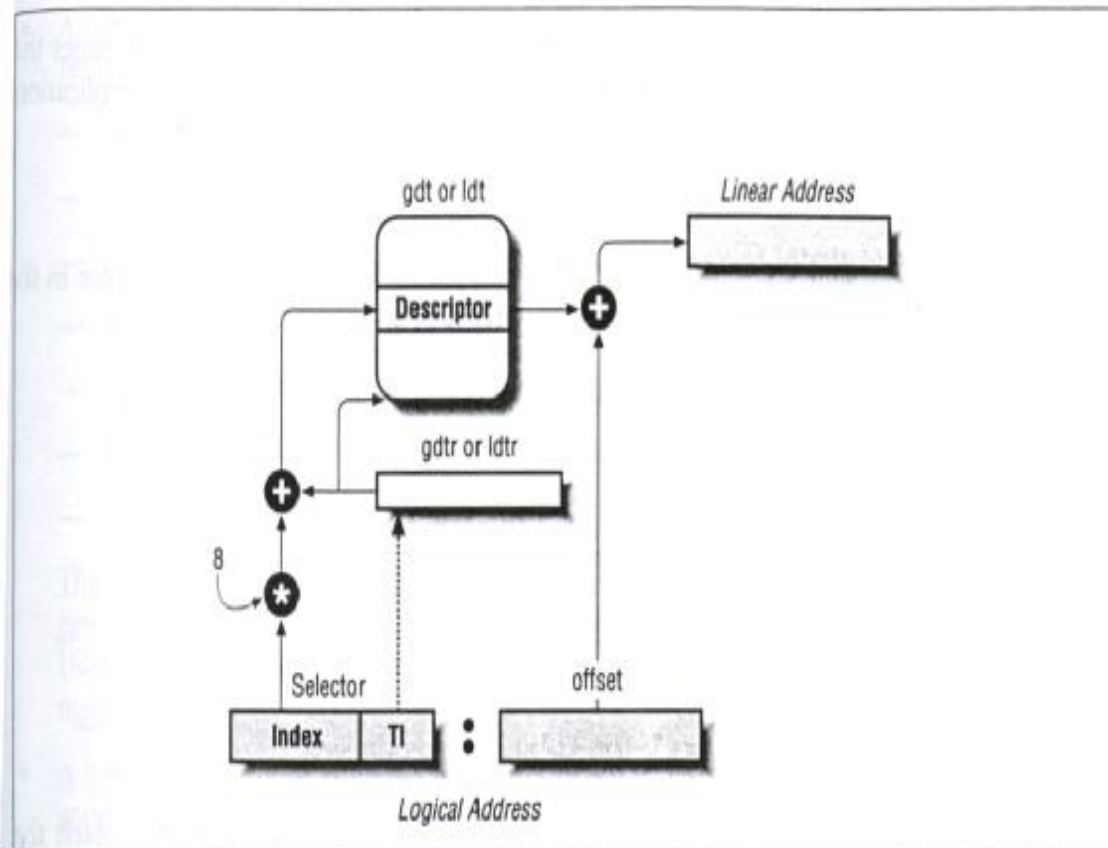


Figure 2-1. Logical address translation

Segmentation in Linux



- | Linux uses segmentation in a very limited way.
- | `__KERNEL_CS`
- | `__KERNEL_DS`
- | `__USER_CS`
- | `__USER_DS`

Paging in x86 (1)

$$2^{10} \times 2^{10} \times 2^{12} = 2^{32}$$
$$1024 \times 1024 \times 4096 = 4G$$

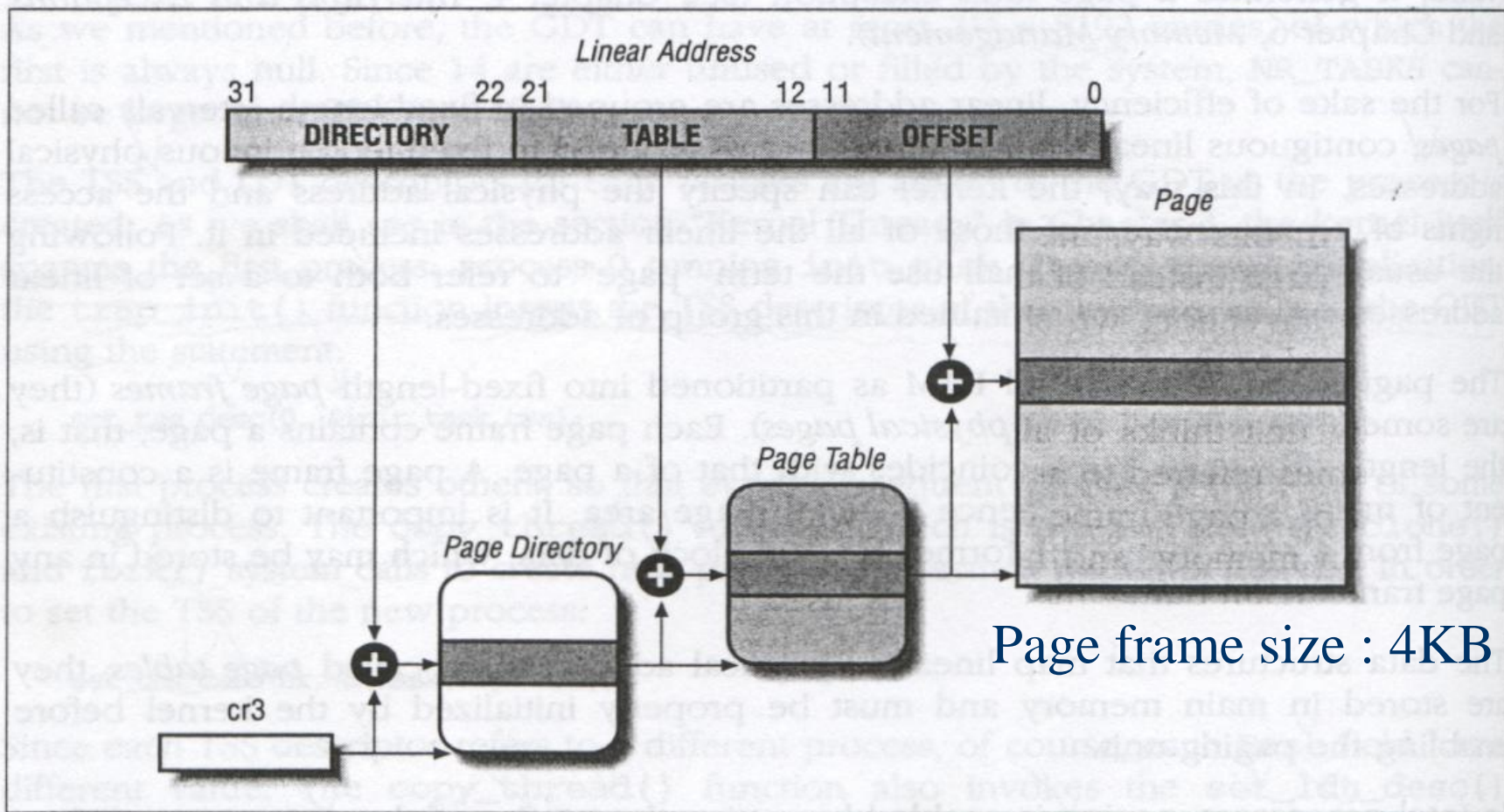


Figure 2-5. Paging by Intel 80x86 processors

Paging in x86 (2)

- I The entries of Page Directories and Page Tables have the same data structure.
 - 20 most significant bits of a page frame physical address (**base address**)
 - **flags** encoded in 12 bits for page protection

- Present flag
- Access flag
- Dirty flag (Only to Page Table entry)
- Read/Write flag
- User/Supervisor flag
- PCD and PWT flag
- Page Size flag (Only to Page Directory entry)

Paging in Linux (1)

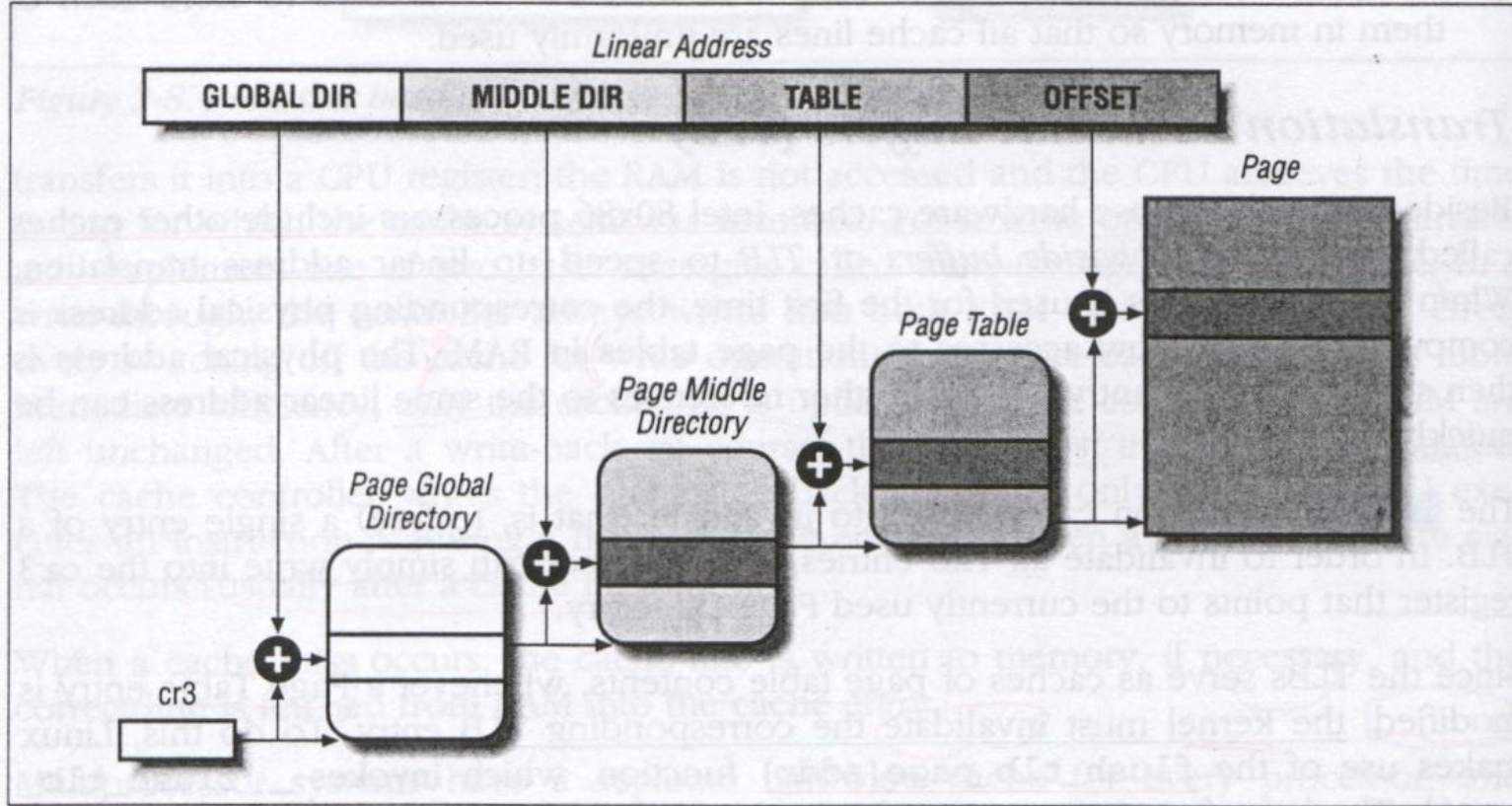


Figure 2-9. The Linux paging model

Paging in Linux (2)

- | Three-level Paging
- | Linux adds “*page middle directory*” between the page directory and the page table.
- | Three level model applied to x86
 - Defining the size of the page middle directory to be one
 - The page middle directory is treated almost interchangeably with the page directory behind macros

Paging in Linux (3)

- I *pgd_t* , *pmd_t* , *pgd_t* are 32-bit data types (unsigned long) for entries
- I The functions and marcos for creating and manipulating entries at each level are defined in several files:
 - include/asm-i386/page.h
 - include/asm-i386/pgtable.h
 - include/asm-i386/pgtable-2level.h
 - include/asm-i386/pgtable-3level.h
 - include/asm-i386/pgalloc.h
 - include/asm-i386/pgalloc-2level.h
 - include/asm-i386/pgalloc-3level.h
 - include/linux/mm.h



Memory Management



Page Frame Management

- | The kernel must keep track of the current status of each page frame.
- | The ***struct page*** (or ***mem_map_t***) is the descriptor of each frame .
- | All the page frame descriptors on the system are included in an array called ***mem_map*** .

Page Frame Management

```
/include/linux/mm.h
typedef struct page {
    struct list_head list;
    struct address_space *mapping;
    unsigned long index;
    struct page *next_hash;
    atomic_t count;
    unsigned long flags;
    struct list_head lru;
    wait_queue_head_t wait;
    struct page **pprev_hash;
    struct buffer_head * buffers;
    void *virtual;
    struct zone_struct *zone;
} mem_map_t;
```

Table 6-1. Flags Describing the Status of a Page Frame

Flag Name	Meaning
PG_decr_after	See the section "The read_swap_cache_async() Function" in, Chapter 16, <i>Swapping: Methods for Freeing Memory</i> .
PG_dirty	Not used.
PG_error	An I/O error occurred while transferring the page.
PG_free_after	See the section "Reading from a Regular File" in Chapter 15, <i>Accessing Regular Files</i> .
PG_DMA	Usable by ISA DMA (see text).
PG_locked	Page cannot be swapped out.
PG_referenced	Page frame has been accessed through the hash table of the page cache (see the section "The Page Cache" in Chapter 14, <i>Disk Caches</i>).
PG_reserved	Page frame reserved to kernel code or unusable.
PG_skip	Used on SPARC/SPARC64 architectures to "skip" some parts of the address space.
PG_Slab	Included in a slab; see the section "Memory Area Management" later in this chapter.
PG_swap_cache	Included in the swap cache; see "The Swap Cache" in Chapter 16.
PG_swap_unlock_after	See the section "The read_swap_cache_async() Function" in Chapter 16.
PG_uptodate	Set after completing a read operation, unless a disk I/O error happened.

Page Fr

Table 6-2. Groups of Flag Values Used to Request Page Frames

Group Name	__GFP_WAIT	__GFP_IO	Priority
GFP_ATOMIC	0	0	__GFP_HIGH
GFP_BUFFER	1	0	__GFP_LOW
GFP_KERNEL	1	1	__GFP_MED
GFP_NFS	1	1	__GFP_HIGH
GFP_USER	1	1	__GFP_LOW

I Request frames :

- **__get_free_pages (gfp_mask , order)**

I Function used to request 2^{order} contiguous page frames

I **gfp_mask** specifies how to look for free page frames.

It consist of : __GFP_WAIT , __GFP_IO , __GFP_DMA ,
__GFP_HIGH , GFP_MED, __GFP_LOW

I Release frames:

- **free_pages (addr, order)**

I Decrease the **count** member of page descriptor

I If **count = 0** , involve **__free_pages_ok ()** to frees a 2^{order} contiguous page frames

The buddy system

- | Low-level page allocation.
- | Efficient strategy for allocating groups of contiguous page frames (deal with *External fragmentation*)
- | Allocates memory from one of a set of disjoint *zones*.
(each represented by a ***struct zone_struct*** – zone_t)
 - Three kind of zones: ZONE_DMA , **ZONE_NORMAL** , ZONE_HIGHMEM
- | With each zone , Free pages are grouped into 10 lists of blocks that contain groups of 1,2,4,16,32,64,128,256 and 512 contiguous page frames, respectively.(a power of 2)

The buddy system

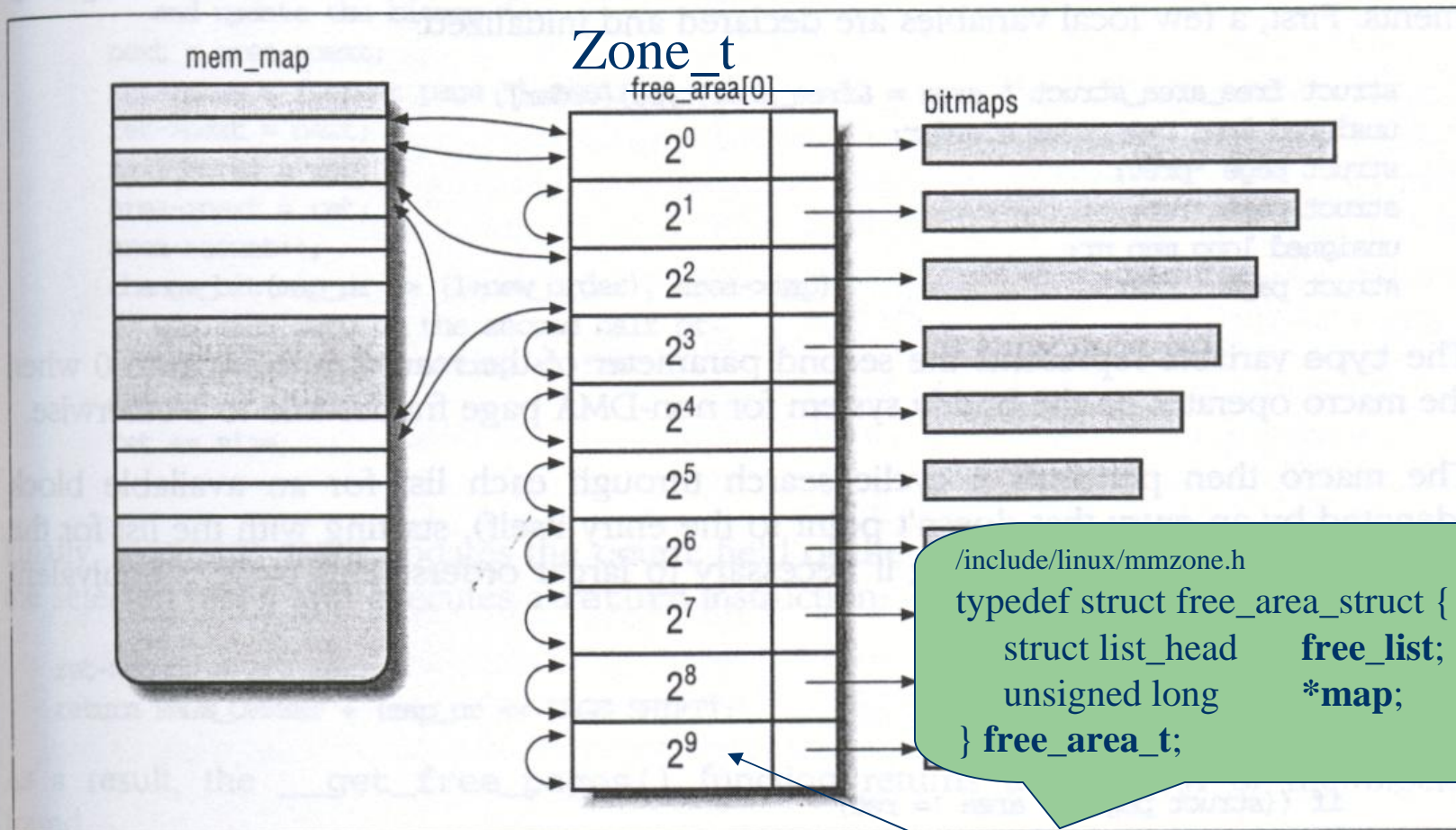


Figure 6-2. Data structures used by the buddy system `free_area_t free_area [9]`

The buddy system

- I Splitting example:
 - The caller wanted 16 pages , but the smallest free region available in the chosen zone was 128 pages

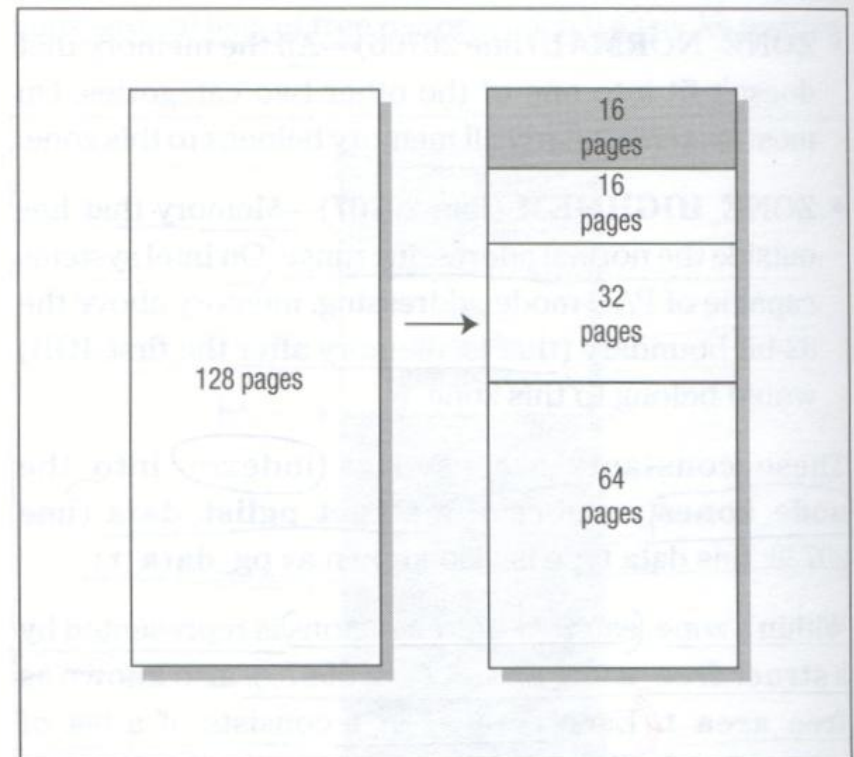


Figure 8.6 Splitting a free region.

The buddy system

- I The *map* field points to a bitmap whose size depends on the number of existing page frames.
- I Each bit of the bitmap describe the status of two buddy blocks.
- I If a bit of the bitmap is equal to **0**, either both buddy blocks of the pair are **free** or both are **busy** ; if it is equal to **1** , exactly one of the block is busy.
- I When both buddies (size 2^k) are free , the kernel treats them as a single free block of size 2^{k+1} .
(Merge)

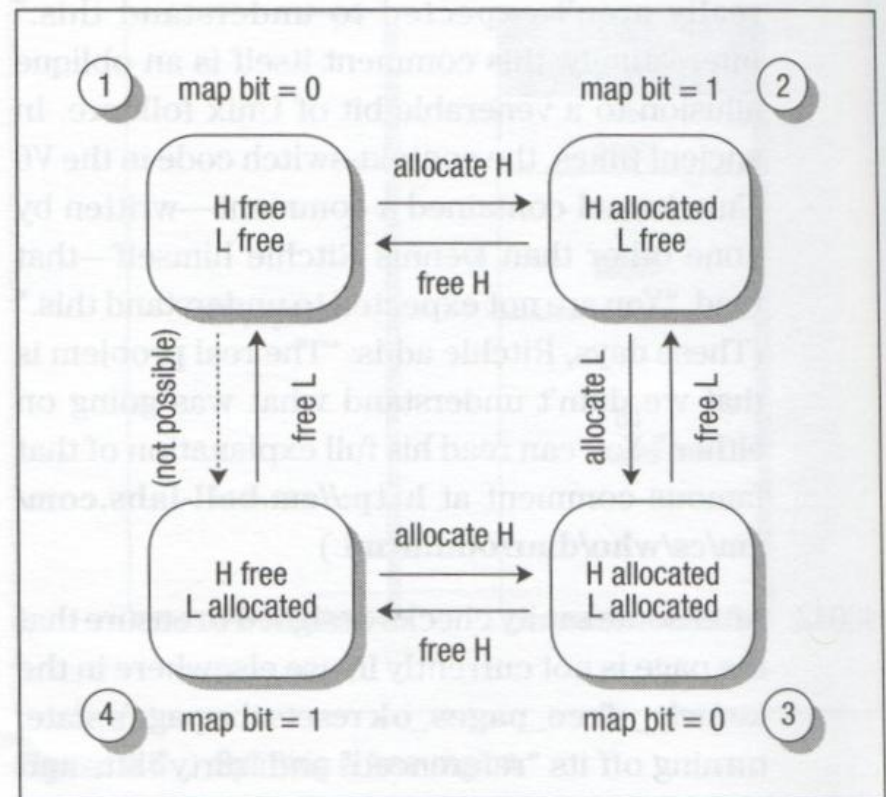


Figure 8.7 Tracking the allocation status of buddies.

The buddy system

`__alloc_pages` (in `/mm/page_alloc.c`)

- I Usually called through a helper function .
 - ex: `__get_free_pages`
- I `struct page * __alloc_pages (unsigned int gfp_mask, unsigned int order, zonelist_t *zonelist)`
- I The function tries to allocates 2^{order} pages from one of the zones in the supplied zone list.
 - Sanity check
 - 2 functions

The buddy system

`__alloc_pages` (in `/mm/page_alloc.c`)

- | Allocating a block of pages with the buddy system can be broken into two basic steps :
 - Finding a large enough free region .
 - Split the free region if it's too large.
- | The function ***rmqueue*** (in `/mm/page_alloc.c`) does the first step.
- | If needed, the function ***expand*** (in `/mm/page_alloc.c`) does the second step.

The buddy system

`__free_pages_ok` (in `/mm/page_alloc.c`)

- | static void `__free_pages_ok` (struct `page` *`page`, unsigned int `order`)
- | `__free_pages_ok` frees a 2^{order} region starting at the given page.
- | If possible, it merges the freed region with its buddy, then merges that larger region with its buddy, and so on.

The Slab Allocator

- | The kernel provides two separate pairs of malloc- and free- like functions
 - ***Kmalloc*** and ***kfree***
 - ***Vmalloc*** and ***vfree*** (latter section)
- | Memory returned by ***kmalloc*** is better purpose such as device drivers , because it's always *physically contiguous*.
- | ***Kmalloc*** is implemented by the Slab Allocator algorithm.

The Slab Allocator- *characteristics*

- | Reduce Internal fragmentation by allocating “objects”.
- | On the top of Buddy system.(The slab allocator reduces the number of calls to the buddy system allocator)
- | The slab allocator groups **objects** into **caches**.
- | Each cache is a “store” of objects of the same type.
- | All objects allocated from a given slab cache have the same size,commonly 2^N bytes.
- | The area of main memory that contains cache is divided into **slabs**,each slab consists of one or more contiguous page frames,that contain both allocated and free objects.

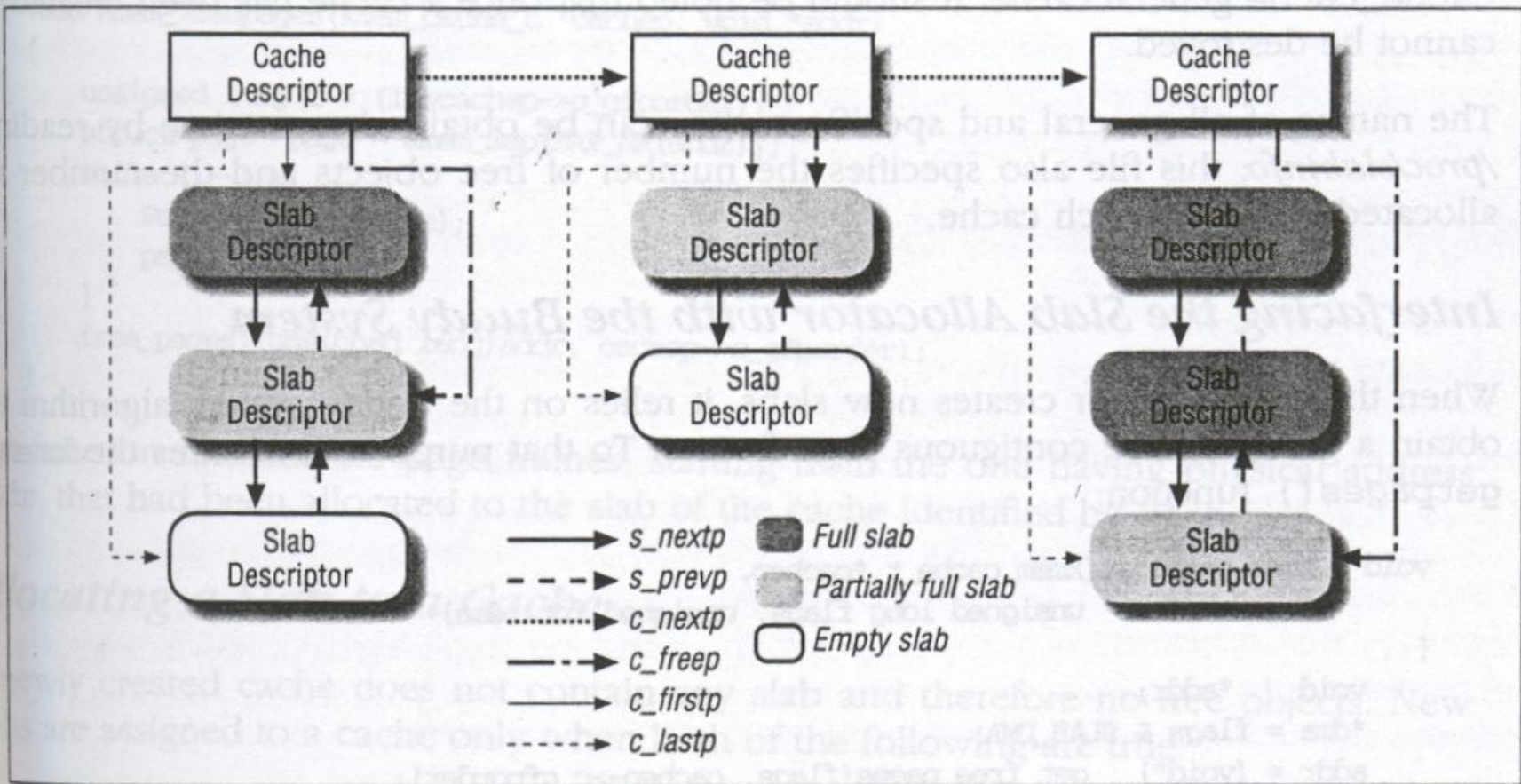


Figure 6-4. Relationships between cache and slab descriptors

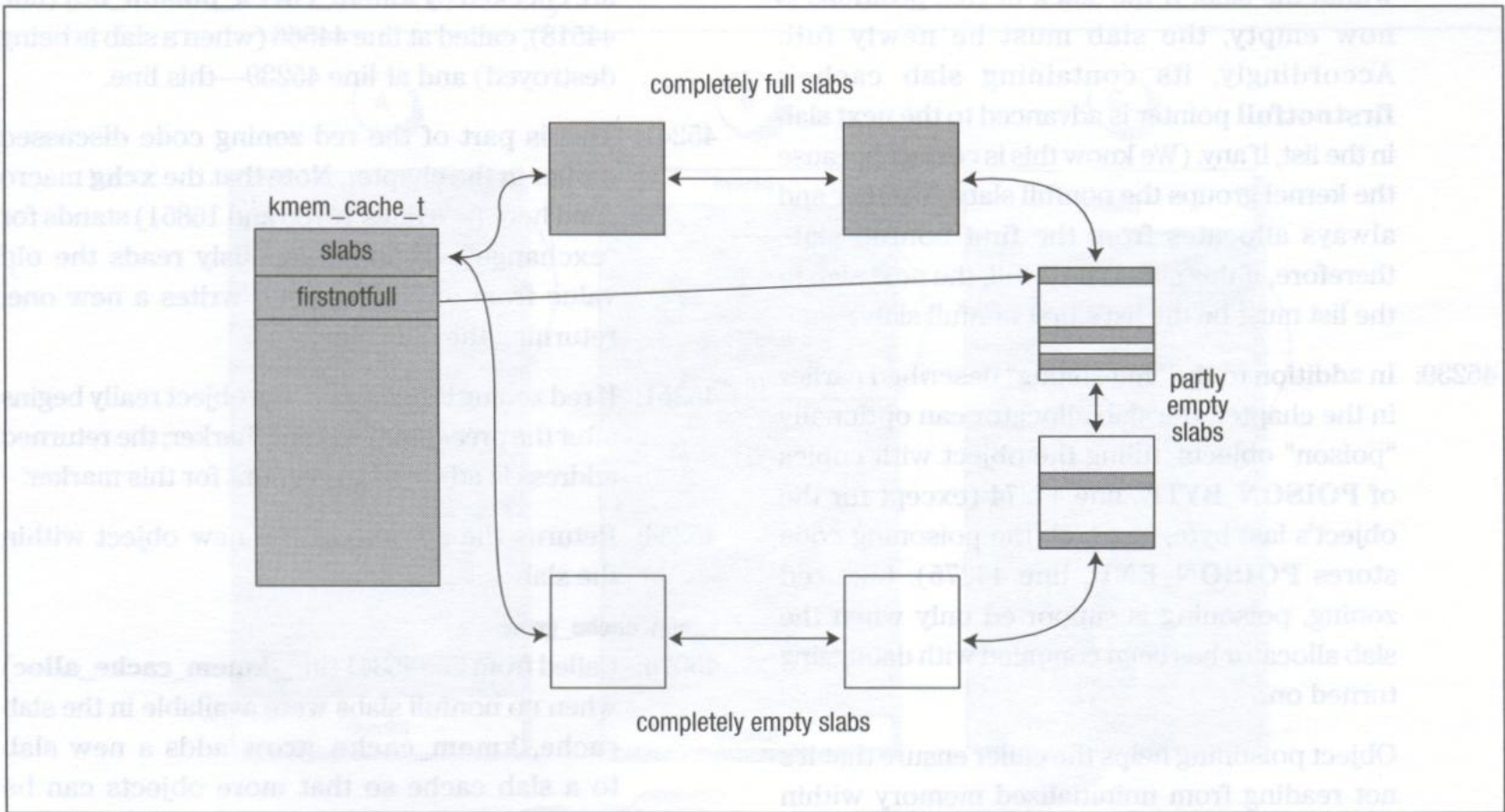


Figure 8.9 A slab cache and its list of slabs.

The Slab Allocator

- I General cache : (`kmem_cache_init()` & `kmem_cache_sizes_init()`)
 - `cache_cache`
 - `cache_slabp`
 - **cache_sizes** (32 bytes to 128KB)
- I Specific cache : (`kmem_cache_create()`)
 - Are used by the remaining parts of the kernel
 - Ex: `kmem_cache_t *vm_area_cache;`
`kmem_cache_t *mm_cache;`

The Slab Allocator

- | ***kmem_cache_create***
 - create a new slab cache
- | ***__kmem_cache_alloc***
 - Called by ***kem_cache_alloc*** & ***kmalloc***
 - Allocate objects
 - ***kmem_cache_grow***
 - | no available nonfull slab → add a new one
 - | Call ***kmem_getpages*** → call ***__getfree_pages*** (buddy system)

vmalloc and vfree

- | ***vmalloc*** and ***vfree*** , allocate and free virtual memory for the kernel's use .
(linear address starting from `PAGE_OFFSET = 0xc0000000` ,the beginning of the forth gigabyte)
- | This range of virtual memory can map to **noncontiguous** physical memory.

vmalloc and vfree

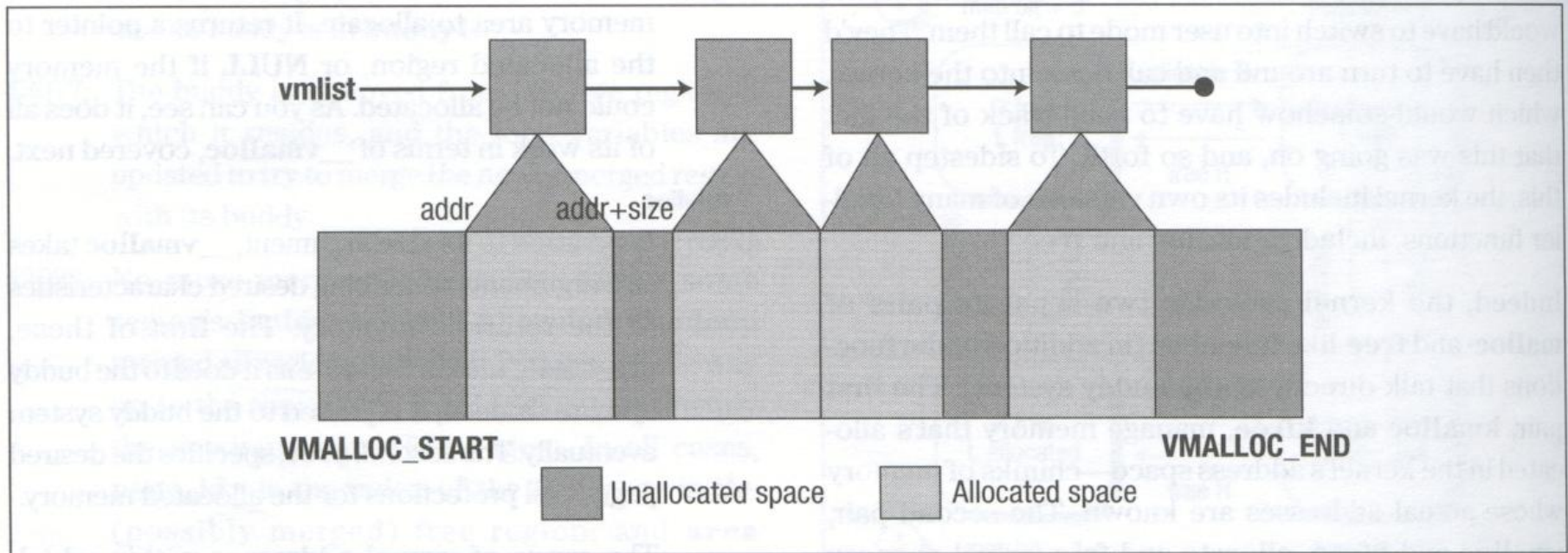


Figure 8.8 The list `vmlist`.

vmalloc and vfree

```
static inline void * vmalloc (unsigned long size)
{
    return __vmalloc(size, GFP_KERNEL);
}

void * __vmalloc (unsigned long size, gfp_mask_t gfp_mask,
                void * addr)
{
    void * addr;
    struct vm_struct *area;
    size = PAGE_ALIGN(size);
    .....
    area = get_vm_area(size, VMALLOC_DEFAULT_FLAGS, gfp_mask, addr);
    addr = area->addr;
    .....
    if ( vmalloc_area_pages(VMALLOC_VMADDR(addr), size, gfp_mask, prot))
    {
        vfree(addr);
        return NULL;
    }
    return addr;
}
}
```

struct **vm_struct** {
 unsigned long flags;
 void * addr;
 unsigned long size;
 struct **vm_struct** * next;
};

Tries to return a free region in the range

Ensures that page-table mapping can be setup (can be noncontiguous)



Process's address space

Data structure

- I Two data structures are important in representing a process's usage of its memory
 - *struct vm_area_struct*
 - *struct mm_struct*

Data structure -

struct vm_area_struct (*/include/linux/mm.h*)

- | The kernel tracks the memory areas used by a process with zero or more ***struct vm_area_structs*** , commonly abbreviated as VMAs.
- | Two VMAs never overlaps.
- | An address can be covered by a VMA even through the kernel hasn't allocated a page in which to store it yet.(....Page fault)
- | Think VMAs as high-level views of what memory areas a process knows about and how these areas are protected.
 - Two VMAs might be discontinuous
 - Two VMAs might have different protections

Data structure - *struct vm_area_struct* (*/include/linux/mm.h*)

```
struct vm_area_struct {  
  
    struct mm_struct * vm_mm;           /* The address space we belong to. */  
    unsigned long vm_start;           /* Our start address within vm_mm. */  
    unsigned long vm_end;           /* The first byte after our end address  
                                     within vm_mm. */  
  
    /* linked list of VM areas per task, sorted by address */  
    struct vm_area_struct * vm_next;  
    unsigned long vm_flags;           /* Flags, listed below. */  
    rb_node_t vm_rb;  
  
    .....  
};
```

Data structure –

struct mm_struct (*/include/linux/sched.h*)

- | The VMAs reserved by a process are all managed by ***struct mm_struct*** .
- | A pointer to a **struct** of this type is in **struct task_struct** – specifically , it's the latter 's **mm** member
- | **struct mm_struct**'s **mmap** member is the linked list of VMAs mentioned earlier.

Data structure – *struct mm_struct* (*/include/linux/sched.h*)

```
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;
    .....
};
```

Data structure

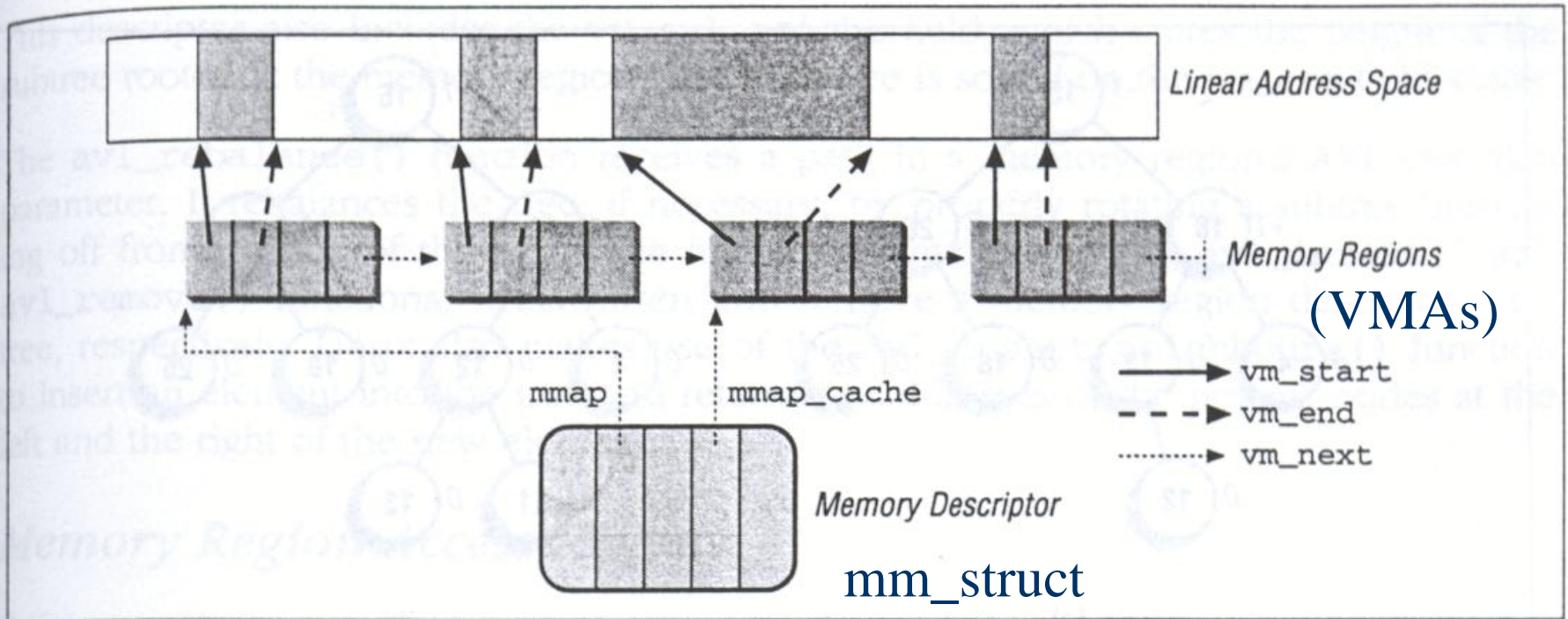


Figure 7-2. Descriptors related to the address space of a process

Operating on VMAs

I *find_vma*

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

I *Find_vma_prev*

- Same as *find_vma*, but also return **a pointer to the previous VMA in *pprev**.

I Search steps:

- Before 2.4.10 : mmap_cache -> list -> AVL tree
- Now : mmap_cache -> Red-Black tree

Operating on VMAs - *find_vma* (1) (*/linux/mm/mmap.c*)

```
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;
        }
    }
}
```

Operating on VMAs - *find_vma (2)* (*/linux/mm/mmap.c*)

```
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;
}
```

Operating on VMAs-

find_vma_prev (/linux/mm/mmap.c)

- I The code is similar to *find_vma* .
 - (Go through the RB tree quickly.)
- I Why not write *find_vma* as a layer over the more general *find_vma_prev* , simply discarding the pointer to the previous VMA?

```
struct vm_area_struct * find_vma(struct mm_struct * mm, unsigned long addr)
{
    struct vm_area_struct * notused ;
    return find_vma_prev(mm , addr , & notused );
}
```

Others

- | Allocating a linear address interval
 - ***do_mmap()*** function creates and initializes a new VMA for current the current process.
 - After a successful allocation , the VMA could be merged with other process (v2.2)
- | Release a linear address interval
 - ***do_munmap()*** function delete a linear address interval from the address space of the current process
 - | Scanning the memory regions
 - | Updating the page table
 - | `ummap_fixup` (4 cases)

References

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