NORTEL

Linux kernel 2.6(4G) Memory Management

Gilbert Wang
Dec 10th, 2009
Rev 0.1

BUSINESS MADE SIMPLE

NORTEL

Agenda

- **Review the MMU of the different architecture**
  - *MIPS architecture*
  - *PowerPC architecture*
  - *i386 series architecture*
- **Linux Memory Management**
  - *Multiple level pages system*
  - *The description about the physical memory*
  - *Boot Memory Allocation*
  - *Buddy system*
  - *Slab system*
  - *Virtual address space & organization*
  - *Memory Mapping*
  - *On-demand page*
  - *Copy on write*
  - *RMAP*

Review the MMU for different architecture (hardware)

What is MMU?

It is the abbreviation for the Memory Management Unit.

It resides in the CPU, which is a controller for being used to manage the virtual memory and physical memory, and also it is responsible for translating the virtual address to the physical address and providing the protecting system in hardware level.

PowerPC MMU

The feature of the MMU about the Classic PowerPC architecture is as follows:

1. *Support for the block address translation (BAT), page address translation (PAT).*

2. *Support for the real mode.*

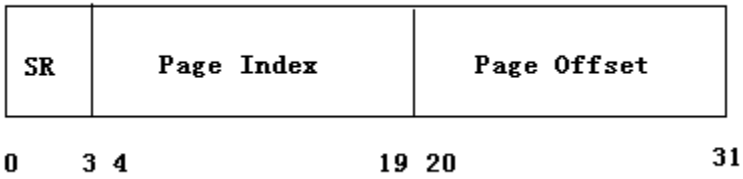
3. *Fixed 4KB pages.*

4. *Segmental memory model.*

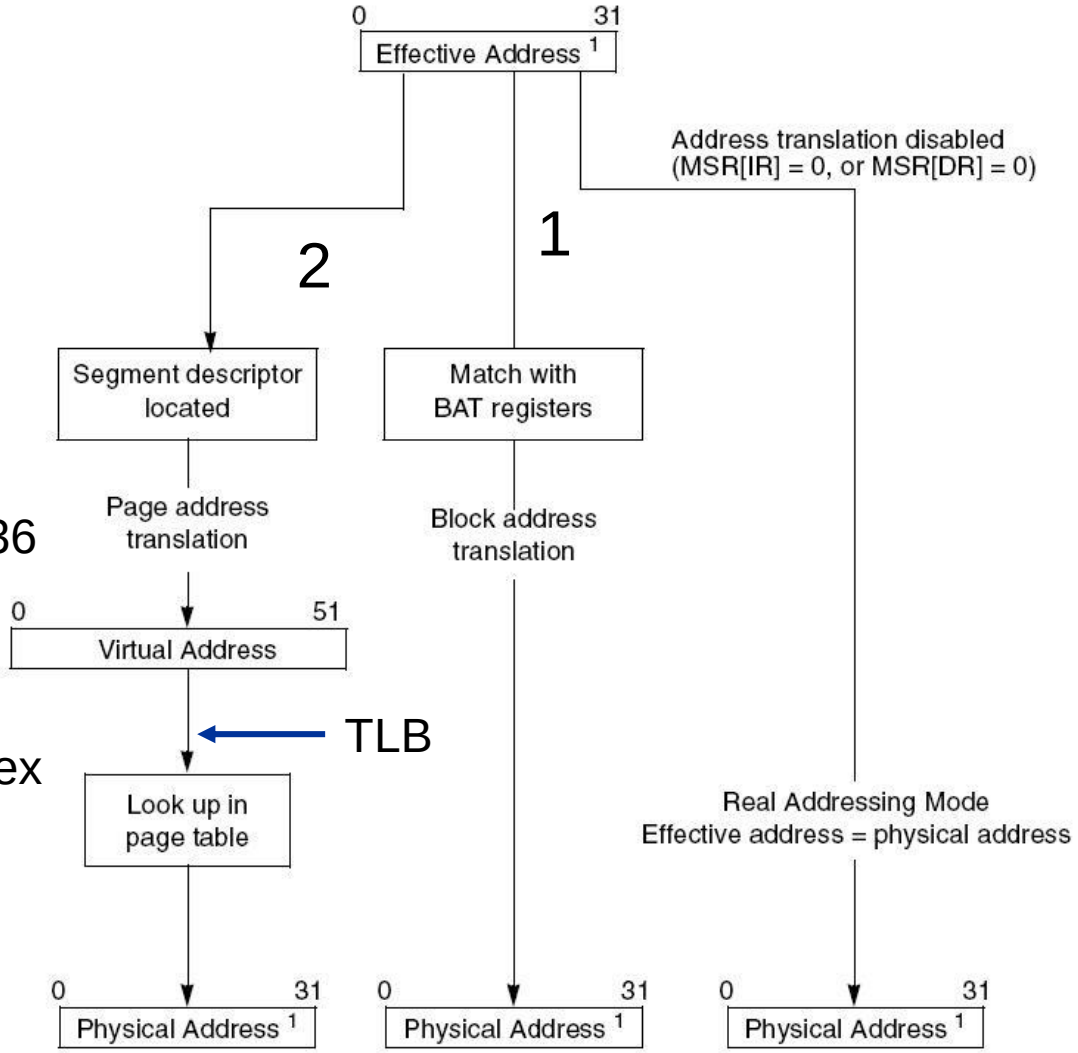
5. *Hardware page address translation.*

6. *Support for the demand page.*

Address structure & translation



Segment number: 16
Segment size: 256M
Page size: 4KB
Page numbers in one segment: 65536
Max size: 4G
VSID: 24bits
Linear address: 52bits
Page Index + VSID = page entry index



The feature of the MMU about the Enhanced PowerPC architecture is as follows:

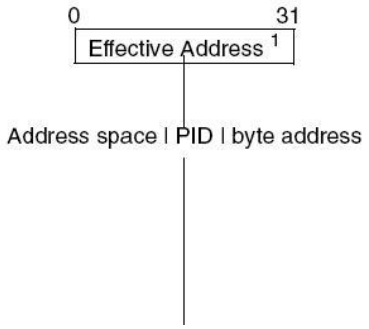
1. Support for the page address translation.

2. Support for the both fixed and variable-sized page translation mechanism.

3. No hardware table hashing and additional features that support management of page translation and protection in the TLB in software. We could access the TLB directly by the software. The exception will be generated, and we must search the destiny page via the software.

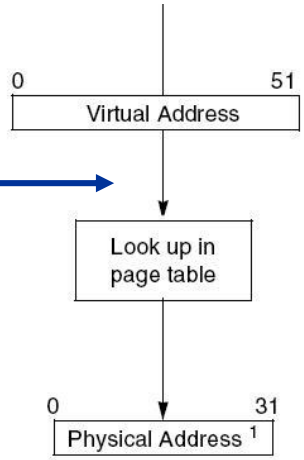
4. Support for the demand page.

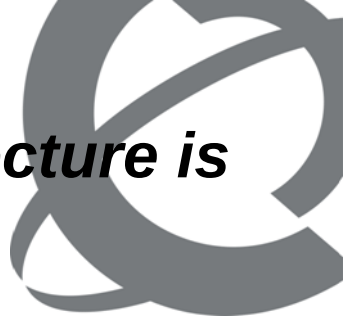
Translation



In Freescale Book E processors, the virtual address is determined by concatenating the address space bit and the process ID (PID) to the effective address.

TLB match





The feature of the MMU about the MIPS architecture is as follows:

1. Support for the page address translation.

2. No real-mode.

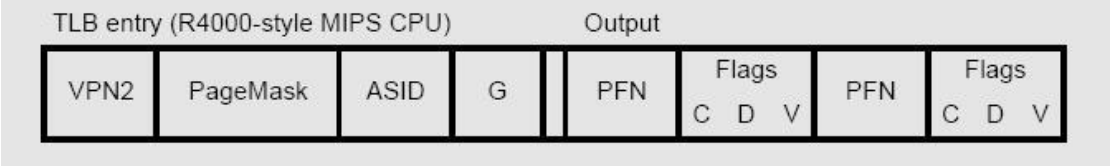
3. Support for the both fixed and variable-sized page translation mechanism. (4KB, 16KB, 64KB, 256KB, 1MB, 4MB and 16MB)

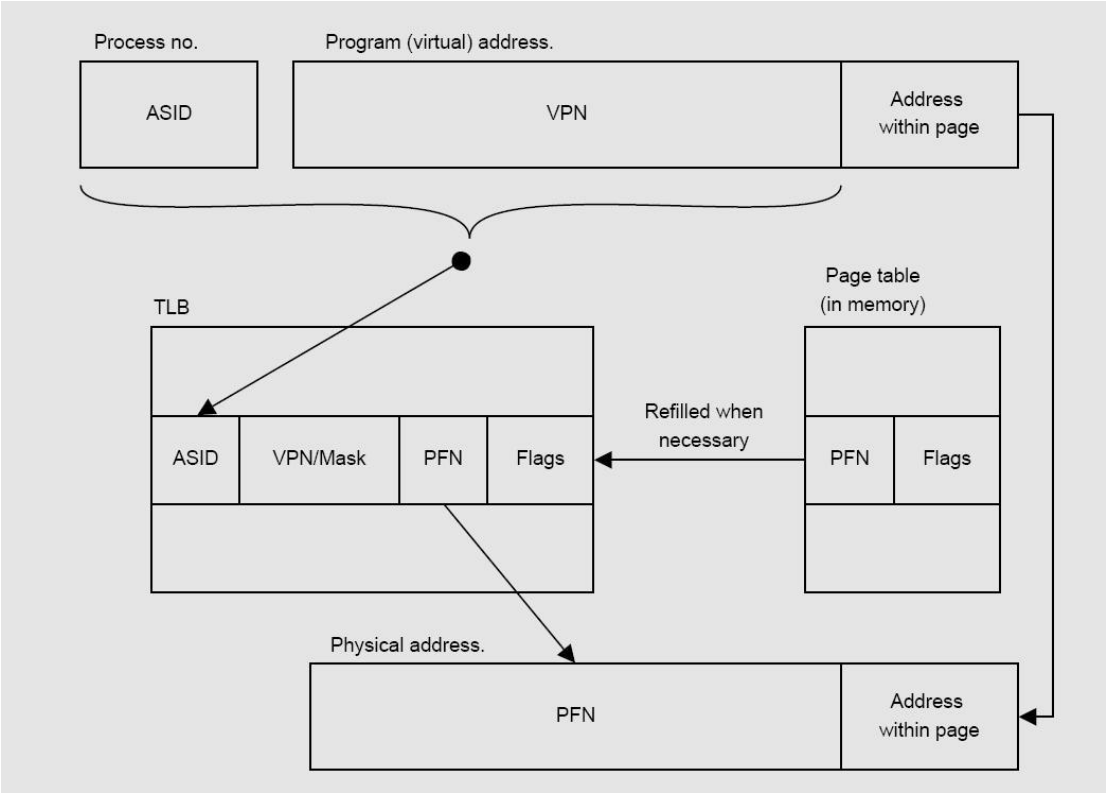
4. No hardware table hashing and additional features that support management of page translation and protection in the TLB in software. We could access the TLB directly by the software. If mismatching in the TLB, "refill" exception will be generated. We must search the destiny page via the software.

5. Each TLB entry is doubled up to map two consecutive VPN to independently specified physical pages.

6. Support for the demand page.

Address structure & translation





More detail:

<<See MIPS Run>>

The feature of the MMU about the X86 architecture is as follows:

1. Support for the segmental-page memory management.

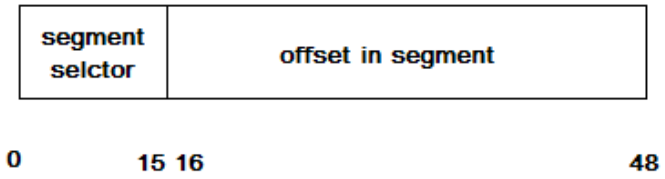
2. Support for the real mode and protected mode.

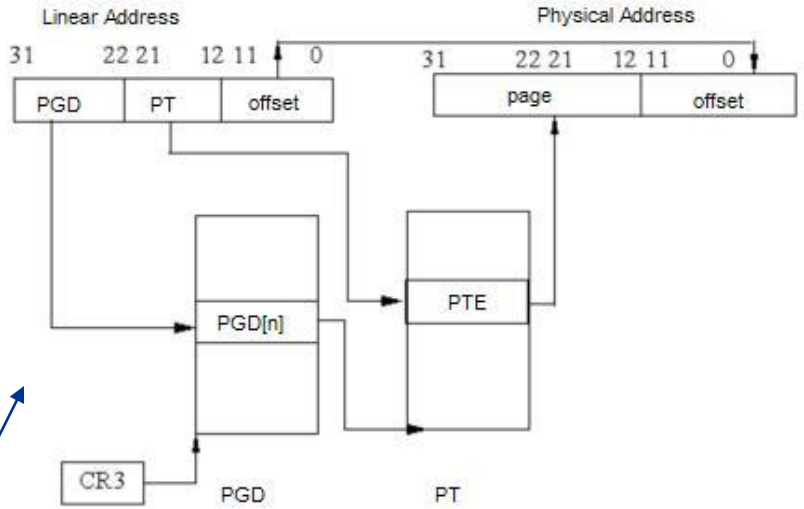
3. Support for the both fixed and variable-sized page translation mechanism. Normally, the page size is 4KB. (4KB, 2MB and 4MB)

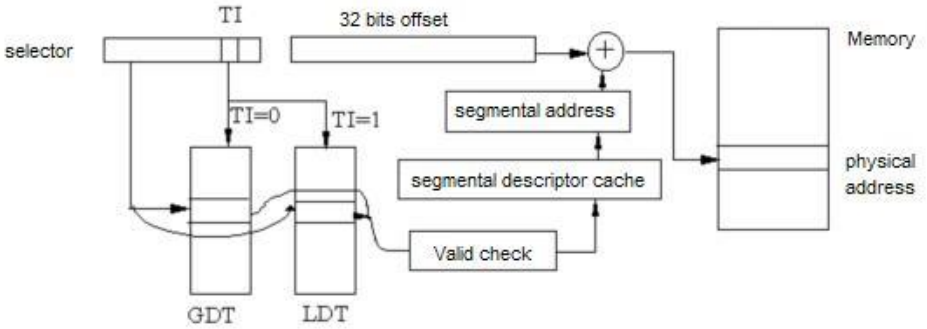
4. Hardware page address translation definition with little architected support for software management.

5. Support for the demand page.

Address structure & translation







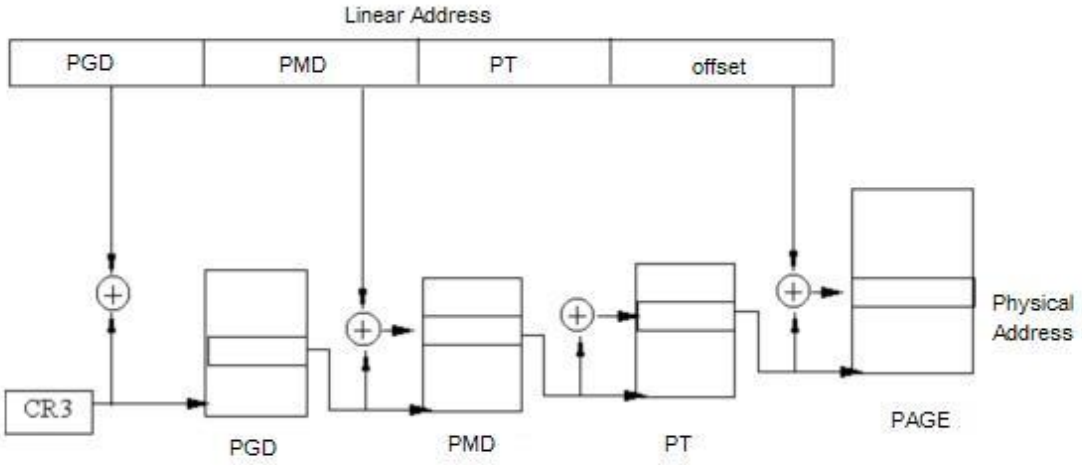
In LTE, the architecture of the OAM/Callp and BH card is MIPS; the MAC card is the PowerPC architecture.

Linux kernel 2.6(4G) Memory Management (software)

Multiple level page

The common paging model is adopted by the Linux 2.6 kernel, 4-level paging.

3-level paging



i386 2-level, PAE enabled 3-level, 64bits 3 or 4-level

PGD->PUD->PMD->PT

```
int __handle_mm_fault(struct mm_struct *mm, struct vm_area_struct *vma,
                      unsigned long address, int write_access)
{
        pgd_t *pgd;
        pud_t *pud;
        pmd_t *pmd;
        pte_t *pte;

        __set_current_state(TASK_RUNNING);

        count_vm_event(PGFAULT);

        if (unlikely(is_vm_hugetlb_page(vma)))
                return hugetlb_fault(mm, vma, address, write_access);

        pgd = pgd_offset(mm, address);
        pud = pud_alloc(mm, pgd, address);
        if (!pud)
                return VM_FAULT_OOM;
        pmd = pmd_alloc(mm, pud, address);
        if (!pmd)
                return VM_FAULT_OOM;
        pte = pte_alloc_map(mm, pmd, address);
        if (!pte)
                return VM_FAULT_OOM;

        return handle_pte_fault(mm, vma, address, pte, pmd, write_access);
}
```

pud=pgd

pmd=pud=pgd

Describe about the physical memory

UMA (Uniform Memory Architecture): The CPU accesses any address in the address space has the same time.
Example: SMP

NUMA (None-Uniform Memory Architecture): The CPU accesses any address in the address space has no the same time.
Example: MPP

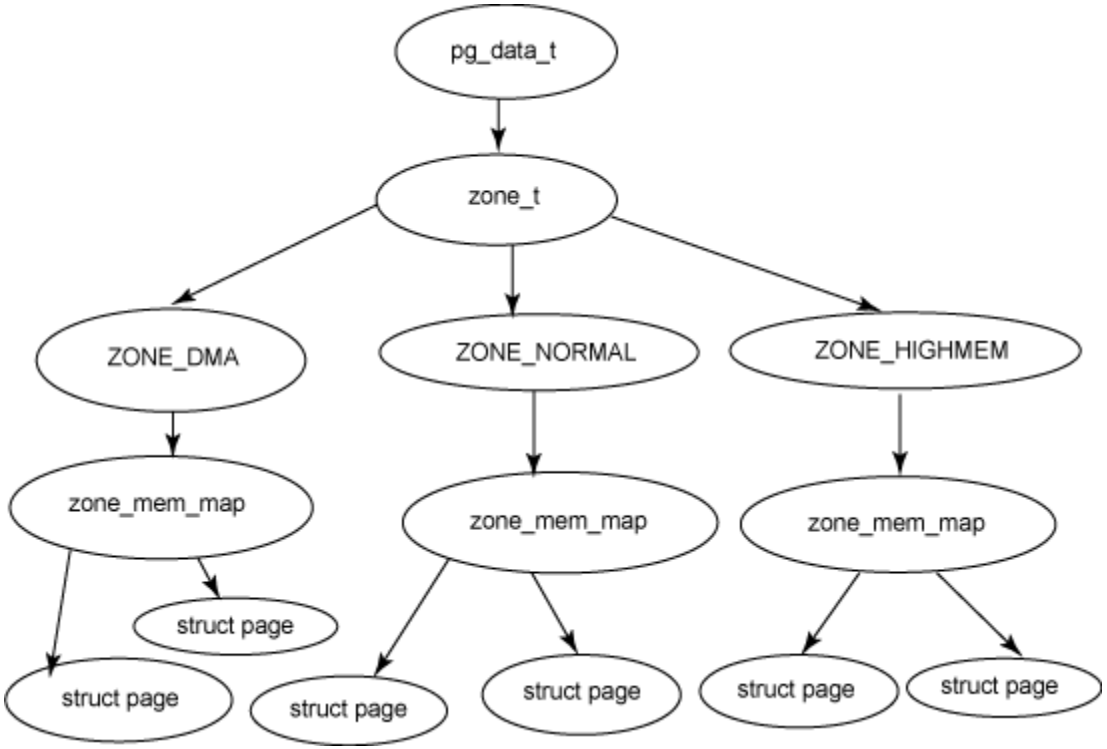
In the Linux

Node : Every UMA is corresponding with a Node. It is consisted of several zones.

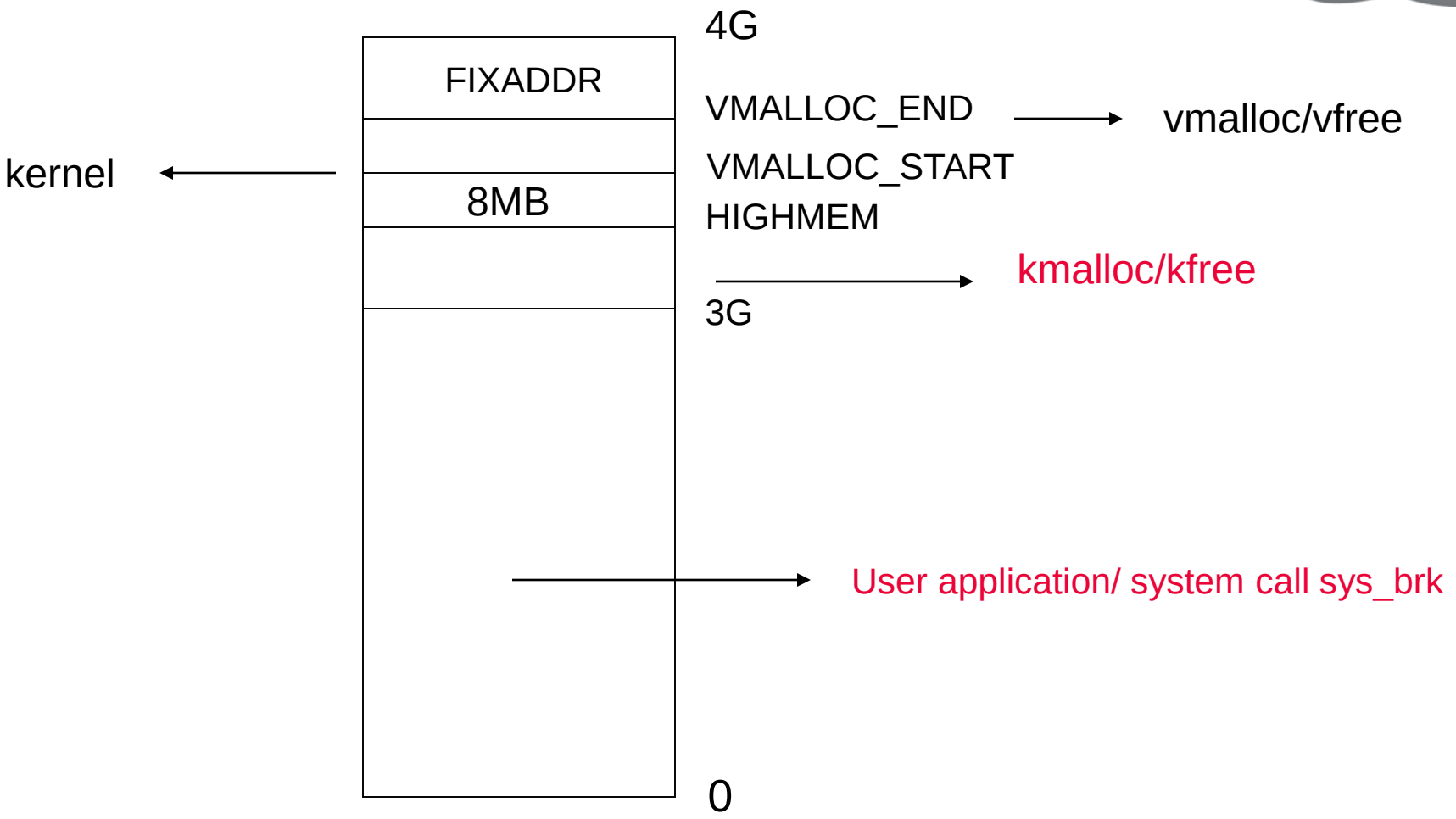
Zone : There are three types of the zone. They are DMA, NORMAL and HIGHMEM.

Page : It is corresponding with a physical page frame, and consist of the Zone.

Relationship



HIGHMEM for x86



The high memory(physical) could not be mapped by the kernel, need page table.

Data structure

struct page ;

typedef struct pglist_data {
.....
} pg_data_t;

struct zone;

Boot Memory allocation

This memory allocator is used in the booting stage.

Method: arch_mem_init

Initializing the memory bit map. (structure boot_mem_map)

```
struct boot_mem_map {
        int nr_map;
        struct boot_mem_map_entry {
                phys_t addr;        /* start of memory segment */
                phys_t size;        /* size of memory segment */
                long type;          /* type of memory segment */
        } map[BOOT_MEM_MAP_MAX];
};

#define BOOT_MEM_RAM            1
#define BOOT_MEM_ROM_DATA       2
#define BOOT_MEM_RESERVED       3
```

The boot allocator will set up the memory bit map.

static inline void bootmem_init(void)

1. Initialize the bootmem_data_t structure. (low memory)

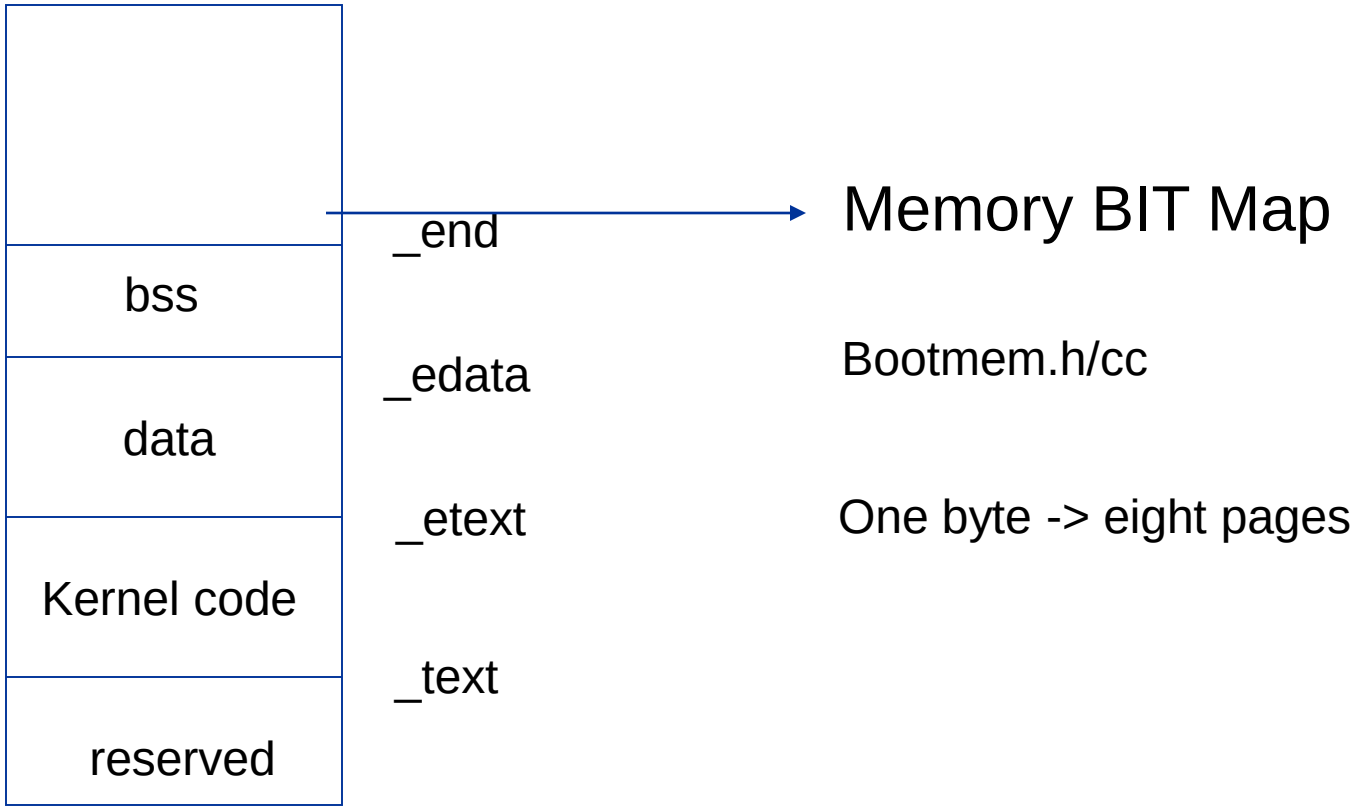
```
bootmap_size = init_bootmem_node(NODE_DATA(0), mapstart, min_low_pfn, max_low_pfn);
```

2. Initialize the bit map.

```
free_bootmem(PFN_PHYS(curr_pfn), PFN_PHYS(size));
```

3. Reserve the memory for the bitmap.

```
reserve_bootmem(PFN_PHYS(first_usable_pfn), bootmap_size);
```



Buddy System

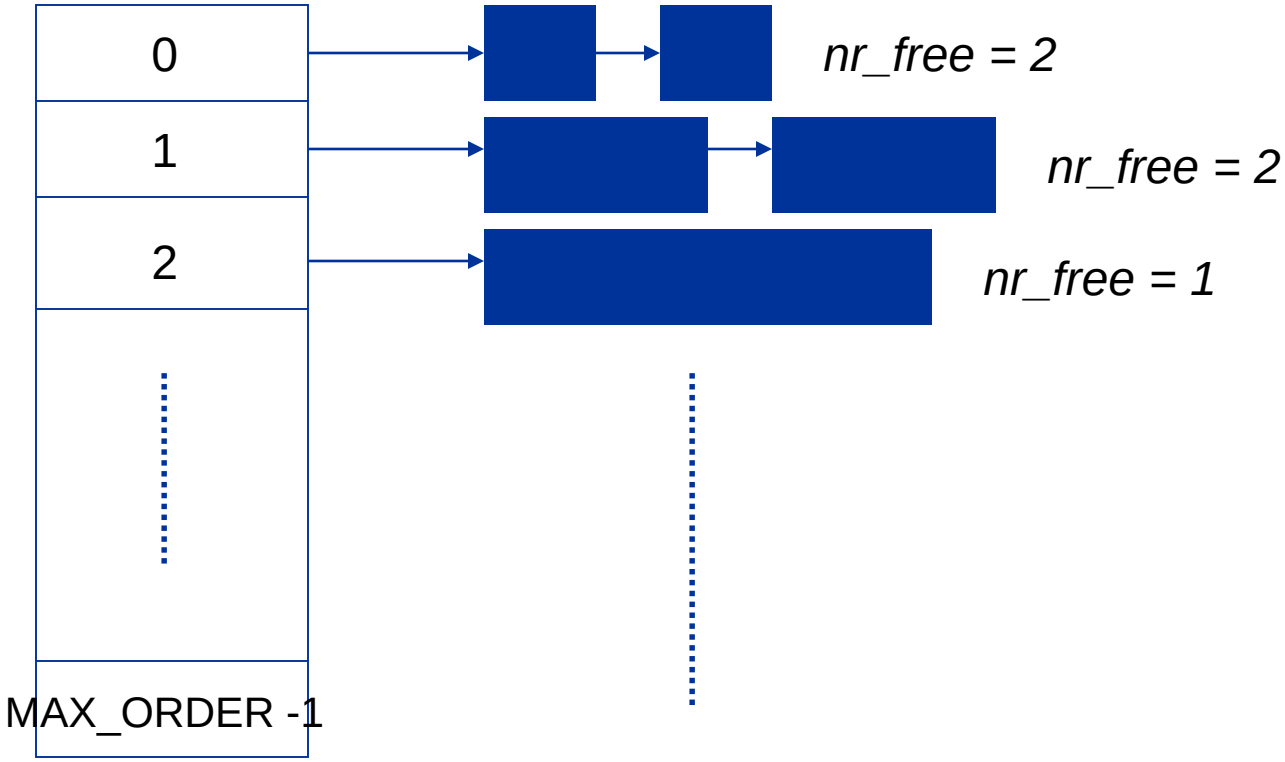
After the memory initialization, the kernel need set up a stable and effective system for allocating and releasing a series of the consecutive pages.

Firstly, the manager will allocate in the virtual address space, and then allocate in the physical address space, at last, the map from the virtual address to physical address will be set up.

The buddy system is used to organize the physical page frame and provide the container for the pages.

Data structure:

```
struct free_area {
        struct list_head    free_list;  //  free pages list
        unsigned long                   nr_free;   //  the number of the free pages
};

struct free_area    free_area[MAX_ORDER];
```



Allocating memory from the buddy system

*struct page * fastcall*
__alloc_pages(gfp_t gfp_mask, unsigned int order,
*struct zonelist *zonelist);*
*static struct page **
get_page_from_freelist(gfp_t gfp_mask, unsigned int order,
*struct zonelist *zonelist, int alloc_flags)*
[note]: if order = 0, then allocating the page from the cold/hot page queue.

*static struct page *__rmqueue(struct zone *zone, unsigned int order);*

*static inline void expand(struct zone *zone, struct page *page,*
*int low, int high, struct free_area *area);*

Releasing the memory

*fastcall void __free_pages(struct page *page, unsigned int order);*
[note]: if order = 0, then releasing the page into hot queue.

static void __free_pages_ok(struct page *page, unsigned int order)

Cold & Hot pages

In every CPU, there are two queues which are the hot/cold page queue.

1. When the releasing the page which size is one page size(4KB), this page will be put into the hot page queue until the queue is up to the high water mark, at this time, these pages will be released to the buddy system.

Releasing the pages:

```
if (pcp->count >= pcp->high) {
        free_pages_bulk(zone, pcp->batch, &pcp->list, 0);
        pcp->count -= pcp->batch;
}
```

2. If you want to apply one page, allocating the pages from the cold or hot page queue which is decided by the caller. If the queue is empty, the system will refill the queue from the buddy system.

Allocating the pages:

```
if (!pcp->count) {
        pcp->count += rmqueue_bulk(zone, 0,
                                pcp->batch, &pcp->list);
        if (unlikely(!pcp->count))
                goto failed;
}
```

User application allocates the memory from heap. (hot pages queue)

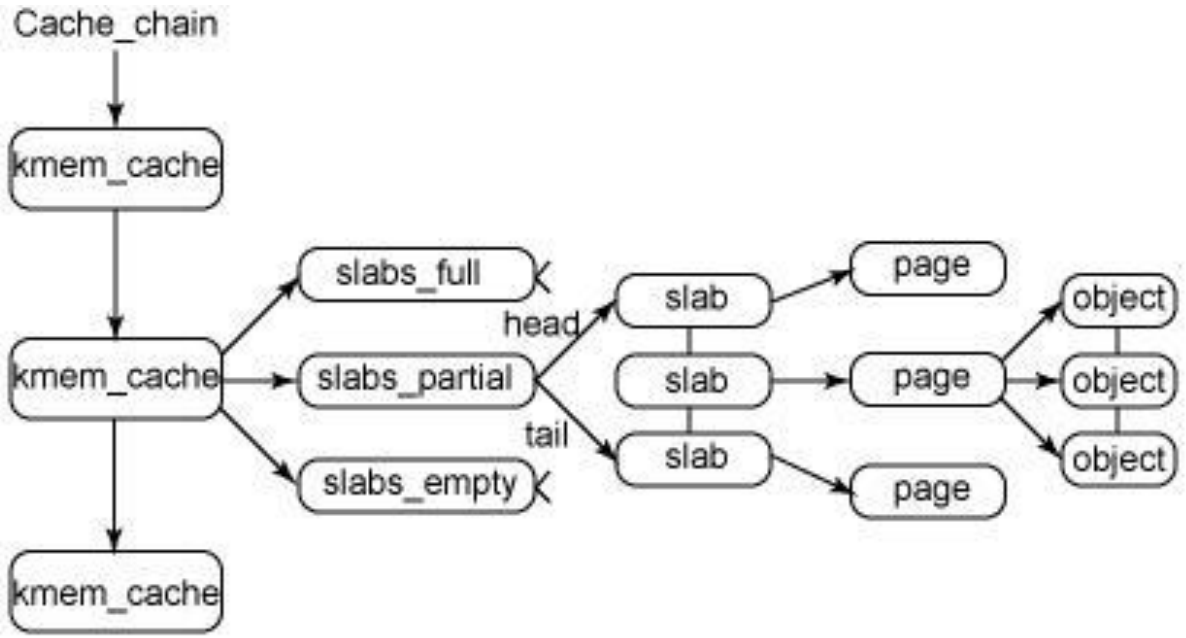
```
static inline int handle_pte_fault(struct mm_struct *mm,
                        struct vm_area_struct *vma, unsigned long address,
                        pte_t *pte, pmd_t *pmd, int write_access)

static int do_anonymous_page(struct mm_struct *mm, struct vm_area_struct *vma,
                        unsigned long address, pte_t *page_table, pmd_t *pmd,
                        int write_access)

static inline struct page *
alloc_zeroed_user_highpage(struct vm_area_struct *vma, unsigned long vaddr)

struct page *
alloc_page_vma(gfp_t gfp, struct vm_area_struct *vma, unsigned long addr)
```

User application maps the file into memory from the disk. (cold pages queue)

```
static inline int handle_pte_fault(struct mm_struct *mm,
                        struct vm_area_struct *vma, unsigned long address,
                        pte_t *pte, pmd_t *pmd, int write_access)

static int do_no_page(struct mm_struct *mm, struct vm_area_struct *vma,
                        unsigned long address, pte_t *page_table, pmd_t *pmd,
                        int write_access)

static struct vm_operations_struct linvfs_file_vm_ops = {
            .nopage             = filemap_nopage,
            .populate           = filemap_populate,
};

static int fastcall page_cache_read(struct file * file, unsigned long offset)

static inline struct page *page_cache_alloc_cold(struct address_space *x)
```

Slab

The SLAB is object-oriented conception for managing the memory in kernel, which is put forward in SUN OS 2.2. It is a series of consecutive pages. It is used to reduce the accessing of the buddy system.



Data structure

struct slab;
struct kmem_cache;

Functions

*static inline void *kmalloc(size_t size, gfp_t flags);*
*void kfree(const void *objp);*

The free memory procedure:

1. Converting the virtual address which will be released into page, and then in terms of the page structure, the cache will be gotten.

2. For every CPU, there is an array cache for optimization. The object will be released into the cache in batch, if the array cache is full, and then, release all the objects from the array cache into slab

```
if (likely(ac->avail < ac->limit)) {
        STATS_INC_FREEHIT(cachep);
        ac->entry[ac->avail++] = objp;
        return;
} else {
        STATS_INC_FREEMISS(cachep);
        cache_flusharray(cachep, ac);
        ac->entry[ac->avail++] = objp;
}
```

Data structure

```
/*
 * struct array_cache
 *
 * Purpose:
 * - LIFO ordering, to hand out cache-warm objects from _alloc
 * - reduce the number of linked list operations
 * - reduce spinlock operations
 *
 * The limit is stored in the per-cpu structure to reduce the data cache
 * footprint.
 *
 */
struct array_cache {
        unsigned int avail;
        unsigned int limit;
        unsigned int batchcount;
        unsigned int touched;
        spinlock_t lock;
        void *entry[0];
};
```

Flushing the cache array

```
if (l3->shared) {
        struct array_cache *shared_array = l3->shared;
        int max = shared_array->limit - shared_array->avail;
        if (max) {
                if (batchcount > max)
                        batchcount = max;
                memcpy(&(shared_array->entry[shared_array->avail]),
                        ac->entry, sizeof(void *) * batchcount);
                shared_array->avail += batchcount;
                goto free_done;
        }
}
free_block(cachep, ac->entry, batchcount, node);
```

The procedure of allocating memory:

1. Getting the object from the array cache.

2. If the array cache is empty, then re-fill the cache from the slab cache.

```
if (likely(ac->avail)) {
   STATS_INC_ALLOCHIT(cachep);
   ac->touched = 1;
   objp = ac->entry[--ac->avail];
} else {
   STATS_INC_ALLOCMISS(cachep);
   objp = cache_alloc_refill(cachep, flags);
}
```

The management about the virtual address space

Data structure

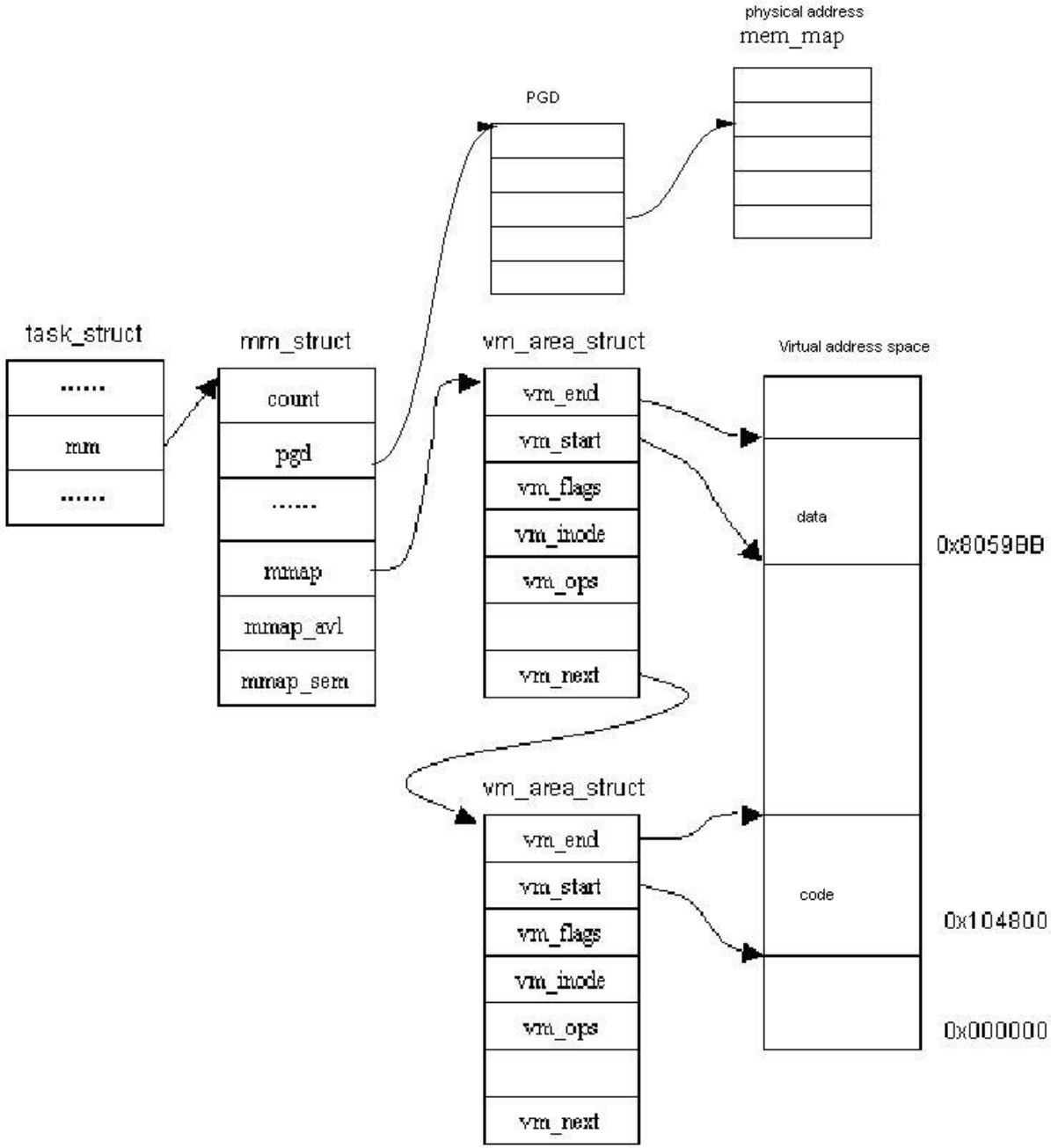
struct mm_struct;
struct vm_area_struct;

1. One process owns one mm_struct structure, which describes the whole user address space.

2. Multiple virtual address zones are consisted of the virtual address space.

3. These virtual address zones are organized by the red-black tree.

4.Putting the last accessing zone into the mmap_cache field.



What is the virtual address zone?

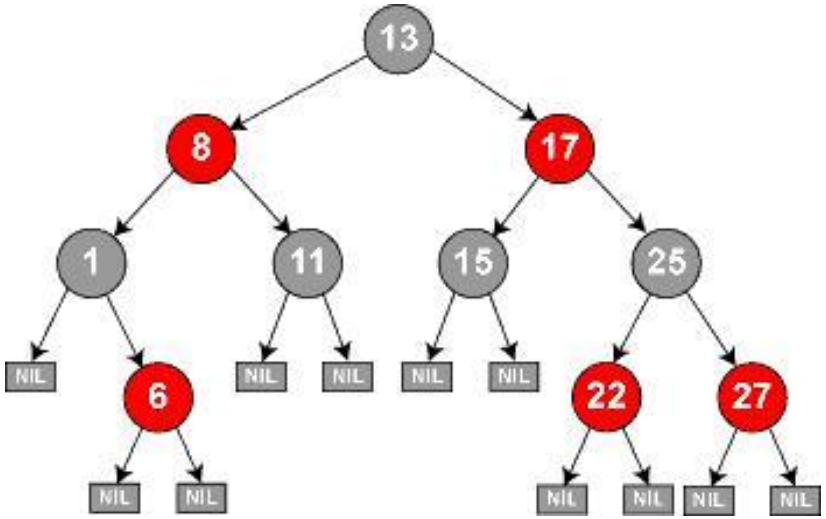
It is used to describe a virtual address segment, including the file mapping, dynamic library mapping and common memory zone.

How does the kernel allocate the dynamic memory?

1. Allocating the space for the process in the virtual address space.

2. If the R/W the space which was allocated before, the physical address space will be allocated.

Red-Black tree



1. The time of searching, deleting and inserting is O (log n).

2. The node is either red or black.

3. The root node is black.

4. If one node is red, the child node must be black.

5. There are same numbers of the black node from any node to the leaf node.

Data structure

```
struct rb_node
{
        struct rb_node *rb_parent;
        int rb_color;
#define RB_RED                  0
#define RB_BLACK        1
        struct rb_node *rb_right;
        struct rb_node *rb_left;
};
```

The implementation is Rbtree.c.

I think it is a good library for reusing via object-oriented encapsulation.

Core functions

*struct vm_area_struct * find_vma(struct mm_struct * mm, unsigned long addr);*

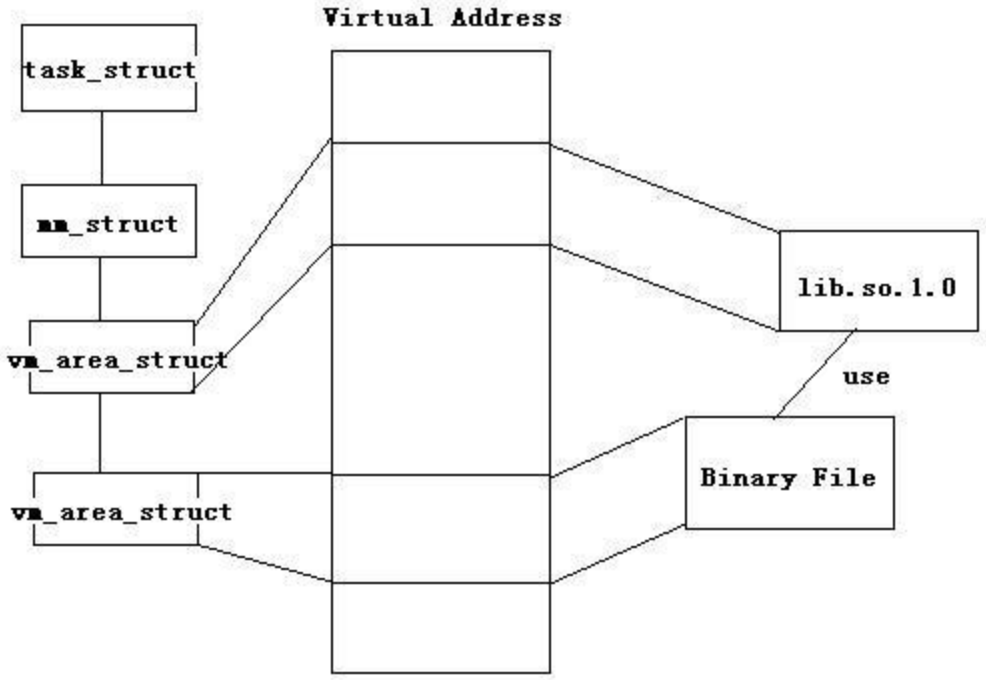
Looking up the first VMA which satisfies addr < vm_end.
1. Checking the cache first. (Cache hit rate is typically around 35%.)
2. Checking the red-black tree. If it is found, it will be saved into the cache.
If it is not found, it will return NULL.

unsigned long do_brk(unsigned long addr, unsigned long len);

1. Creating the vm_struct structure with kmem_cache_alloc from slab.
2. Initializing the structure.
3. Inserting the vm into red-black tree.
[note]: which is called by system call sys_brk.

Memory Mapping

Conception: When the binary image begins to execute, the executive file and the dynamic libraries which are used by the binary image will be linked into the virtual address space of the process. The real physical memory will not be allocated until the program is executed. The binary image will be described by the vm_area_struct structure.



Core functions

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

1. Looking up the vm_area_struct in the red-black tree of the current process.

```
munmap_back:
    vma = find_vma_prepare(mm, addr, &prev, &rb_link, &rb_parent);
    if (vma && vma->vm_start < addr + len) {
        if (do_munmap(mm, addr, len))
            return -ENOMEM;
        goto munmap_back;
    }
```

2. Allocating the vm_area_struct structure by the slab allocator.

```
vma = kmem_cache_alloc(vm_area_cachep, SLAB_KERNEL);
if (!vma) {
        error = -ENOMEM;
        goto unacct_error;
}
memset(vma, 0, sizeof(*vma));
vma->vm_mm = mm;
vma->vm_start = addr;
vma->vm_end = addr + len;
vma->vm_flags = vm_flags;
vma->vm_page_prot = protection_map[vm_flags & 0x0f];
vma->vm_pgoff = pgoff;
```

3. Linking the vm_area_struct into the red-black tree.

```
vma_link(mm, vma, prev, rb_link, rb_parent);
```

On-Demand page

It is a dynamic memory allocating technique. The allocating of the physical page will be delayed as late as it could be. The page fault will be raised.
Advantage: Enhancing the throughput of the system, and improving the efficiency of the memory.
Disadvantage: Take more cycle of the CPU for handling the page fault exception.

Core functions

*int __handle_mm_fault(struct mm_struct *mm, struct vm_area_struct *vma, unsigned long address, int write_access)；*

Allocating the page table, page directory item and middle page directory.

```
pgd = pgd_offset(mm, address);
pud = pud_alloc(mm, pgd, address);
if (!pud)
        return VM_FAULT_OOM;
pmd = pmd_alloc(mm, pud, address);
if (!pmd)
        return VM_FAULT_OOM;
pte = pte_alloc_map(mm, pmd, address);
if (!pte)
        return VM_FAULT_OOM;
```

```
static inline int handle_pte_fault(struct mm_struct *mm,
                struct vm_area_struct *vma, unsigned long address,
                pte_t *pte, pmd_t *pmd, int write_access);


    if (!pte_present(entry)) {
            if (pte_none(entry)) {
                    if (!vma->vm_ops || !vma->vm_ops->nopage)
                            return do_anonymous_page(mm, vma, address,
                                    pte, pmd, write_access);
                    return do_no_page(mm, vma, address,
                                    pte, pmd, write_access);
            }

static int do_no_page(struct mm_struct *mm, struct vm_area_struct *vma,
                unsigned long address, pte_t *page_table, pmd_t *pmd,
                int write_access);


 new_page = vma->vm_ops->nopage(vma, address & PAGE_MASK, &ret);
```

COW (Copy on write)

•When creating the child process, the parent process will shared the address space with the child process, that means the page directory table and page table will be shared.

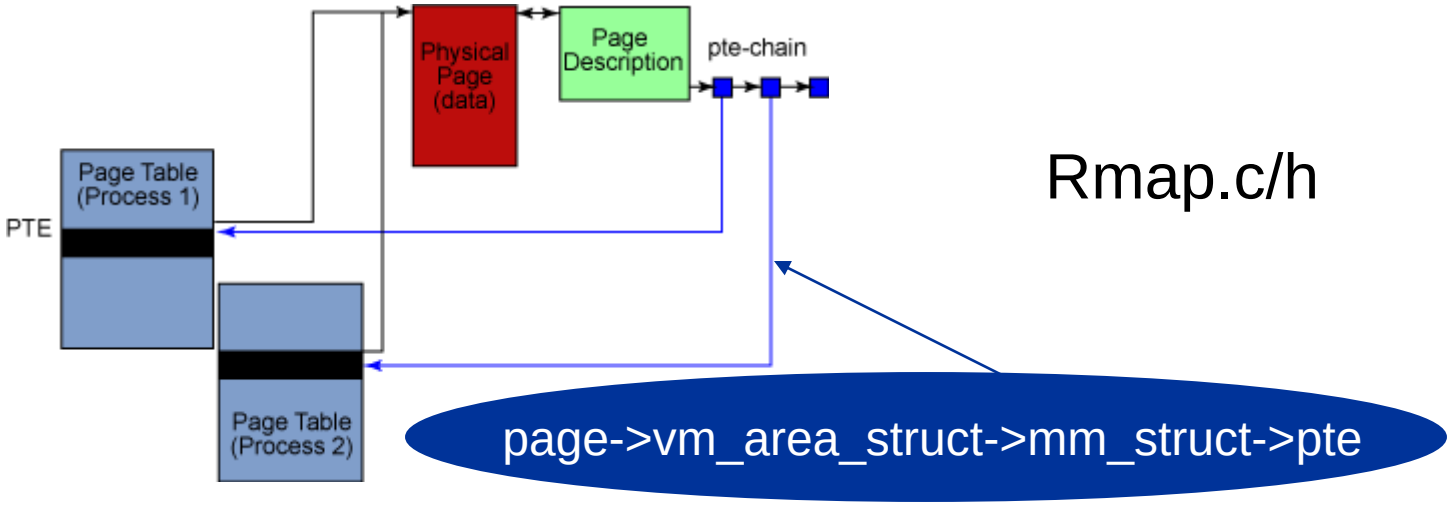
•Once the page is shared with different process, it could not be modified. If the page is tried to modify, the page fault will be generated, the COW will occur.

•The new page will be generated, and the content in the old page will be copied into the new page, and open the writing permission for the new page.

•When one process tries to write the old page, if the process is the only owner of the page, the writing permission will be opened.

RMAP

The kswapd kernel thread will switch some pages into swap disk if the memory is not enough for current running process by the swapping algorithm. At this time, the kswapd will decide the page table of these pages which will be switched into disk. So it will traverse all the tables of the processes. It is time consuming.



Rmap.c/h

Advantage: the performance of the swapping is enhanced.

Disadvantage: More memory will be used**.**

vmscan.c -> shrink_page_list

```
if (page_mapped(page) && mapping) {
        switch (try_to_unmap(page, 0)) {
        case SWAP_FAIL:
                goto activate_locked;
        case SWAP_AGAIN:
                goto keep_locked;
        case SWAP_SUCCESS:
                ; /* try to free the page below */
        }
}
```

Rmap.c->try_to_unmap_anon

```
anon_vma = page_lock_anon_vma(page);

list_for_each_entry(vma, &anon_vma->head, anon_vma_node) {
        ret = try_to_unmap_one(page, vma, migration);
        if (ret == SWAP_FAIL || !page_mapped(page))
                break;
}
```

Rmap.c->try_to_unmap_one

```
struct mm_struct *mm = vma->vm_mm;
pte = page_check_address(page, mm, address, &ptl);
page_remove_rmap(page, vma);
```

```
void page_add_anon_rmap(struct page *page,
        struct vm_area_struct *vma, unsigned long address)
{
        if (atomic_inc_and_test(&page->_mapcount))
                __page_set_anon_rmap(page, vma, address);
        /* else checking page index and mapping is racy */
}


static void __page_set_anon_rmap(struct page *page,
        struct vm_area_struct *vma, unsigned long address)
{
        struct anon_vma *anon_vma = vma->anon_vma;

        BUG_ON(!anon_vma);
        anon_vma = (void *) anon_vma + PAGE_MAPPING_ANON;
        page->mapping = (struct address_space *) anon_vma;

        page->index = linear_page_index(vma, address);

        /*
         * nr_mapped state can be updated without turning off
         * interrupts because it is not modified via interrupt.
         */
        __inc_zone_page_state(page, NR_ANON_PAGES);
}
```

Q & A

NORTEL