Linux内核编程13期
内存管理

主讲：王利涛

01 什么是内存管理？

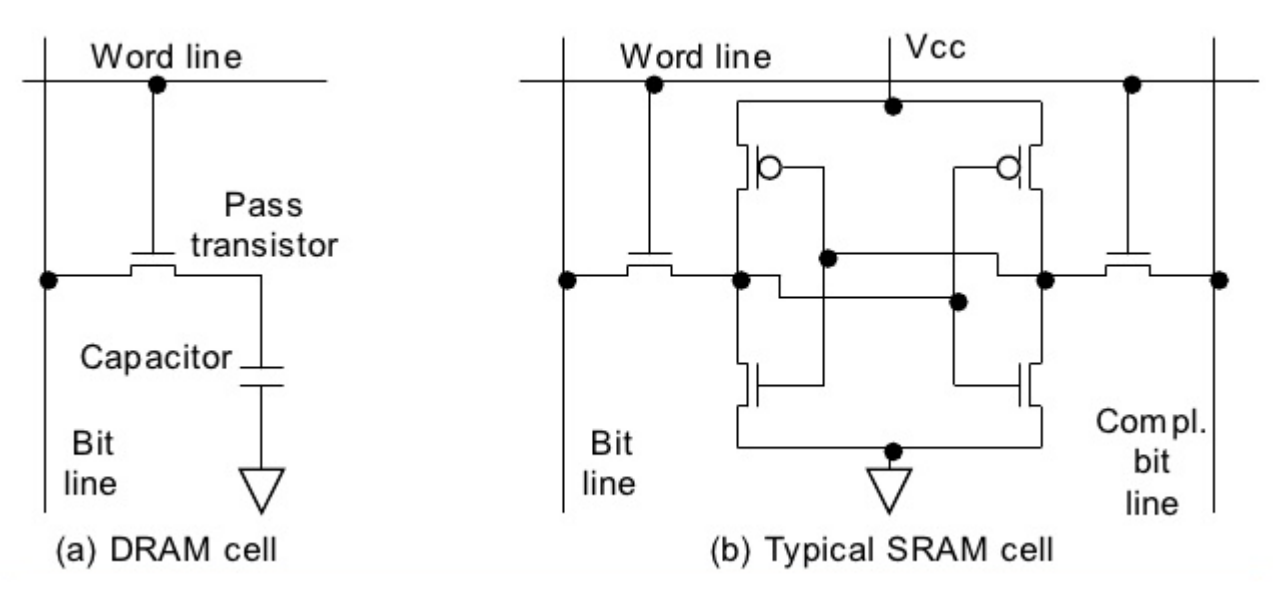
主讲：王利涛

- 内存管理相关
  - 为什么要分为虚拟内存和物理内存？
  - 用户空间、内核空间
  - 物理地址、虚拟地址
  - 页表是什么？谁在维护？存在哪里？什么格式？
  - MMU、TLB
  - 映射：文件映射、匿名映射、IO内存映射
  - 驱动如何申请内存？
  - 面试：缺页中断、伙伴系统…

- 本期课程主要内容
  - 物理内存管理：zone、page、伙伴系统
  - 虚拟内存管理
  - MMU、页表、TLB
  - 内存申请与释放接口
  - 映射机制底层实现
  - 预期收获：
    - » 物理内存、虚拟内存的划分
    - » 深入理解页表、地址转换、映射
    - » 学会使用内核提供的接口申请内存
    - » 构建一个完整的内存管理框架

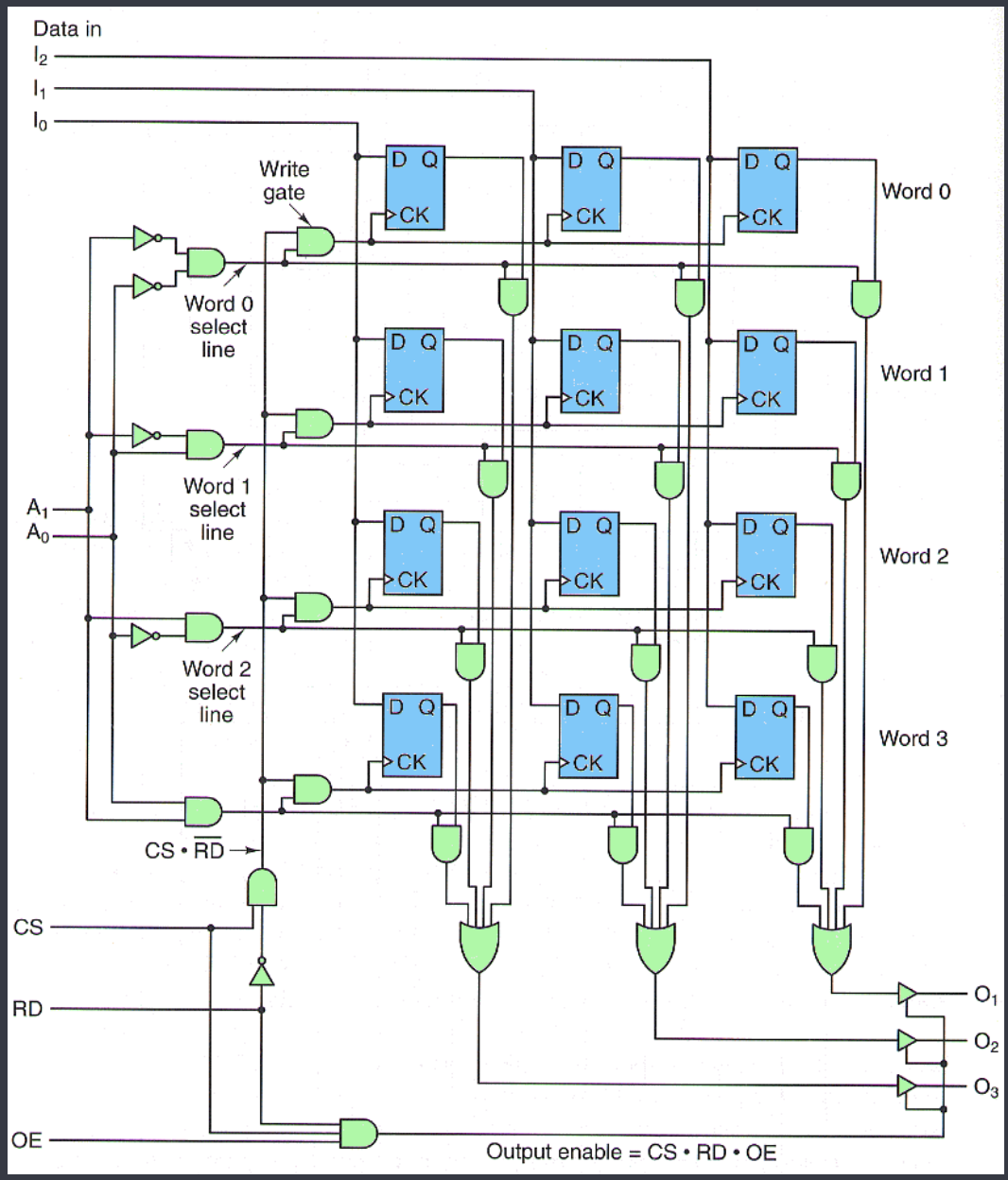
02 内存的硬件电路与接口

- 内存硬件实现
  - Data Latch
  - D触发器
    - 使用D触发器构建寄存器

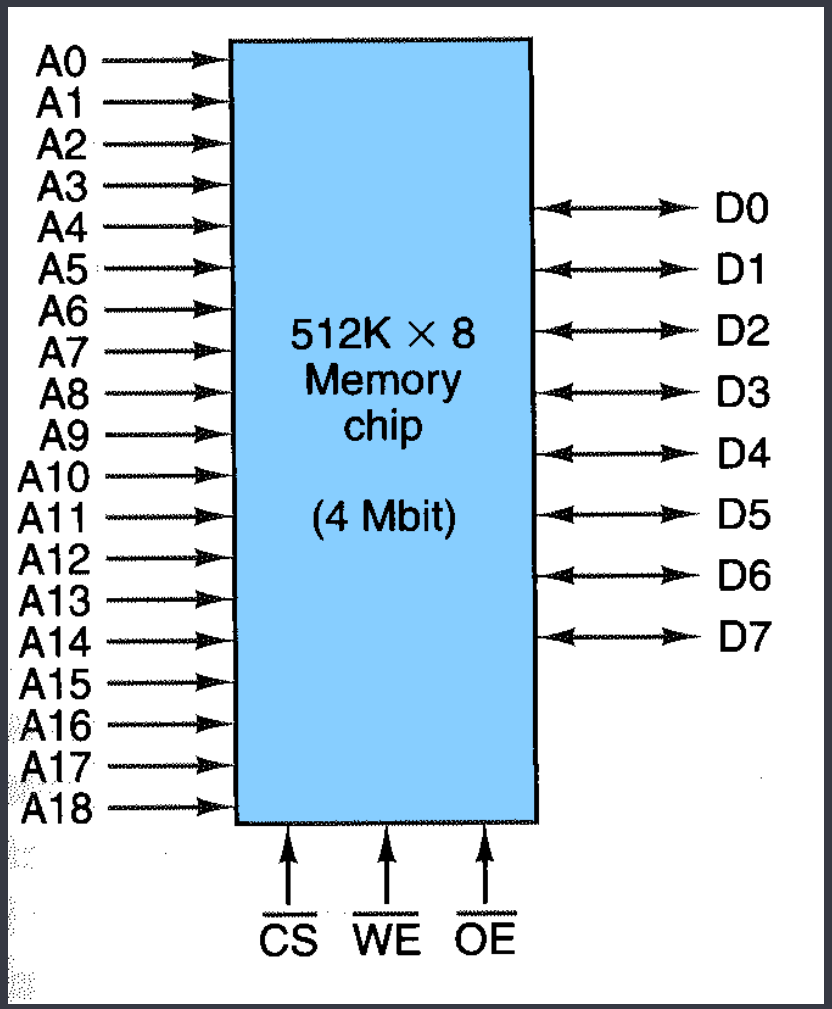


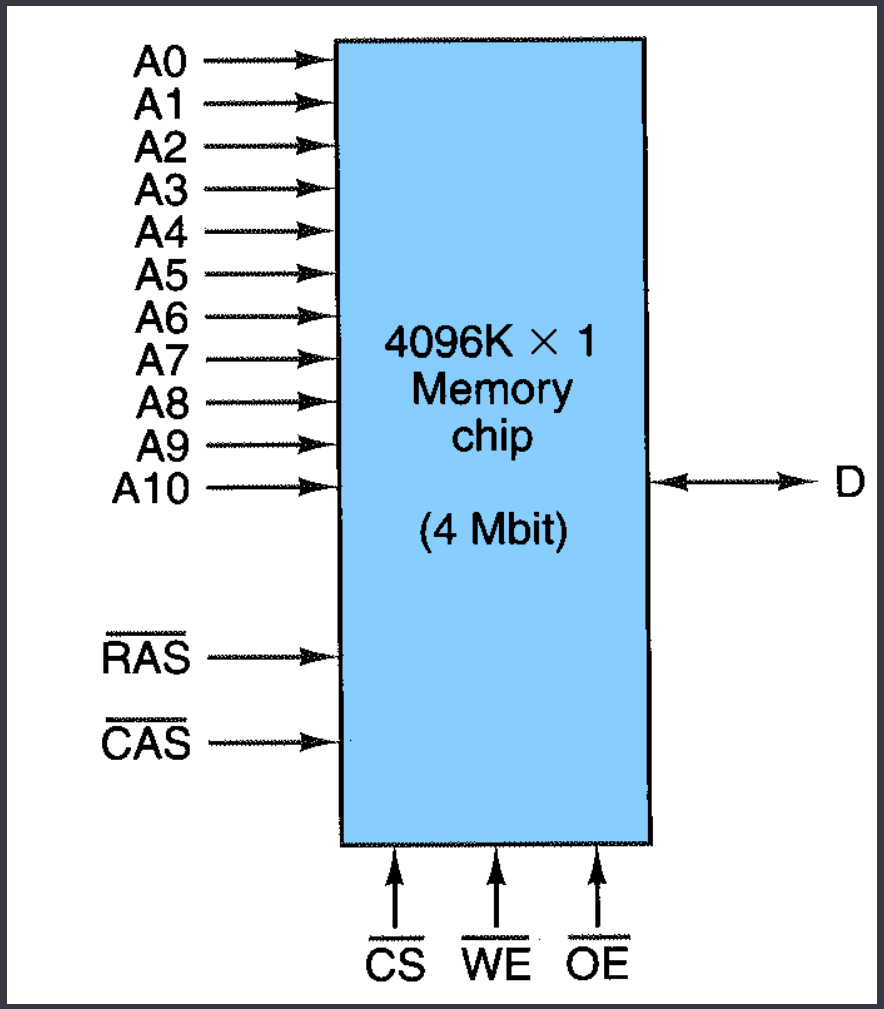
- 使用D触发器构建内存

参考教程：
《深入理解RISC处理器架构》
09 寄存器电路的实现
10 DRAM电路的实现分析

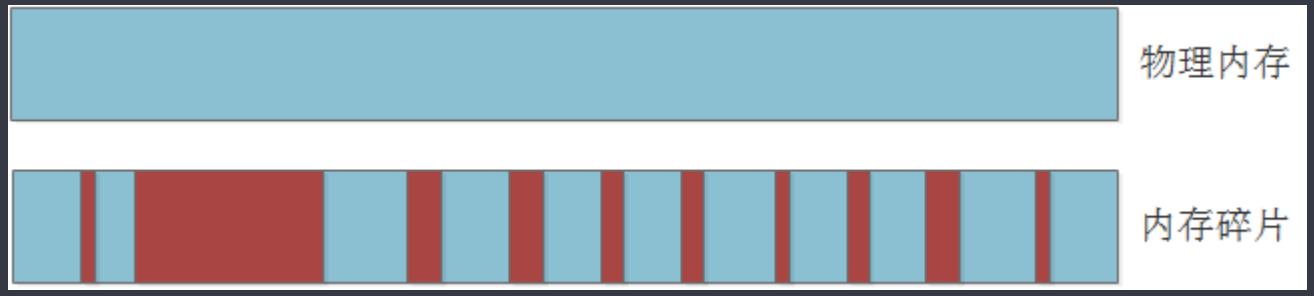


- SRAM & DDR SDRAM



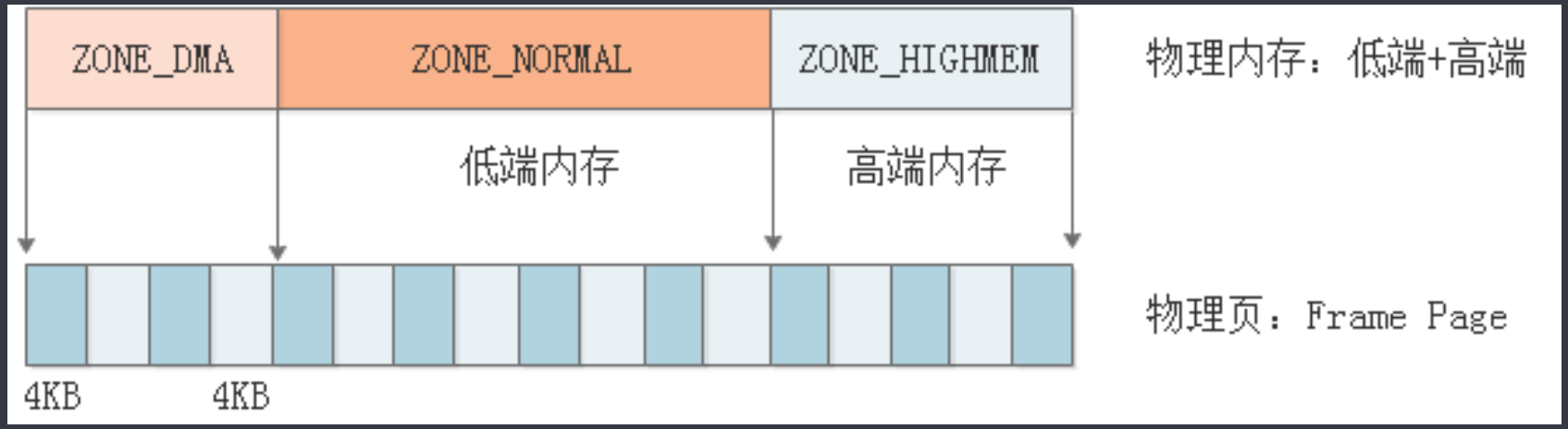


- 内存泄漏
  - 广义的内存泄漏
  - 狭义的内存泄漏
  - 解决之道
    - 裸机环境
    - RTOS平台
    - Linux/Android平台



03 物理内存管理
page、zone和node

- 物理内存管理
  - 页： struct page
  - 分区： struct zone
  - 内存节点： struct node



- 物理页帧： struct page
  - 定义头文件：include/linux/mm_types.h
  - 每个物理页帧(page frame)使用结构体page表示
  - 结构体struct page核心成员分析
  - 思考：
    - 物理页帧和 struct page之间的关系
    - 物理页帧号(page frame number)和物理地址的关系
    - struct page存储在哪里？
    - 全局变量：mem_map指针

```
#define __page_to_pfn(page) ((unsigned long) ((page) - mem_map) + ARCH_PFN_OFFSET)
#define __pfn_to_page(pfn)  (mem_map + ((pfn) - ARCH_PFN_OFFSET))
```

- 内存区域：struct zone
  - 定义：include/linux/mmzone.h
  - 结构体struct zone核心成员解读
  - 初始化：zone_sizes_init

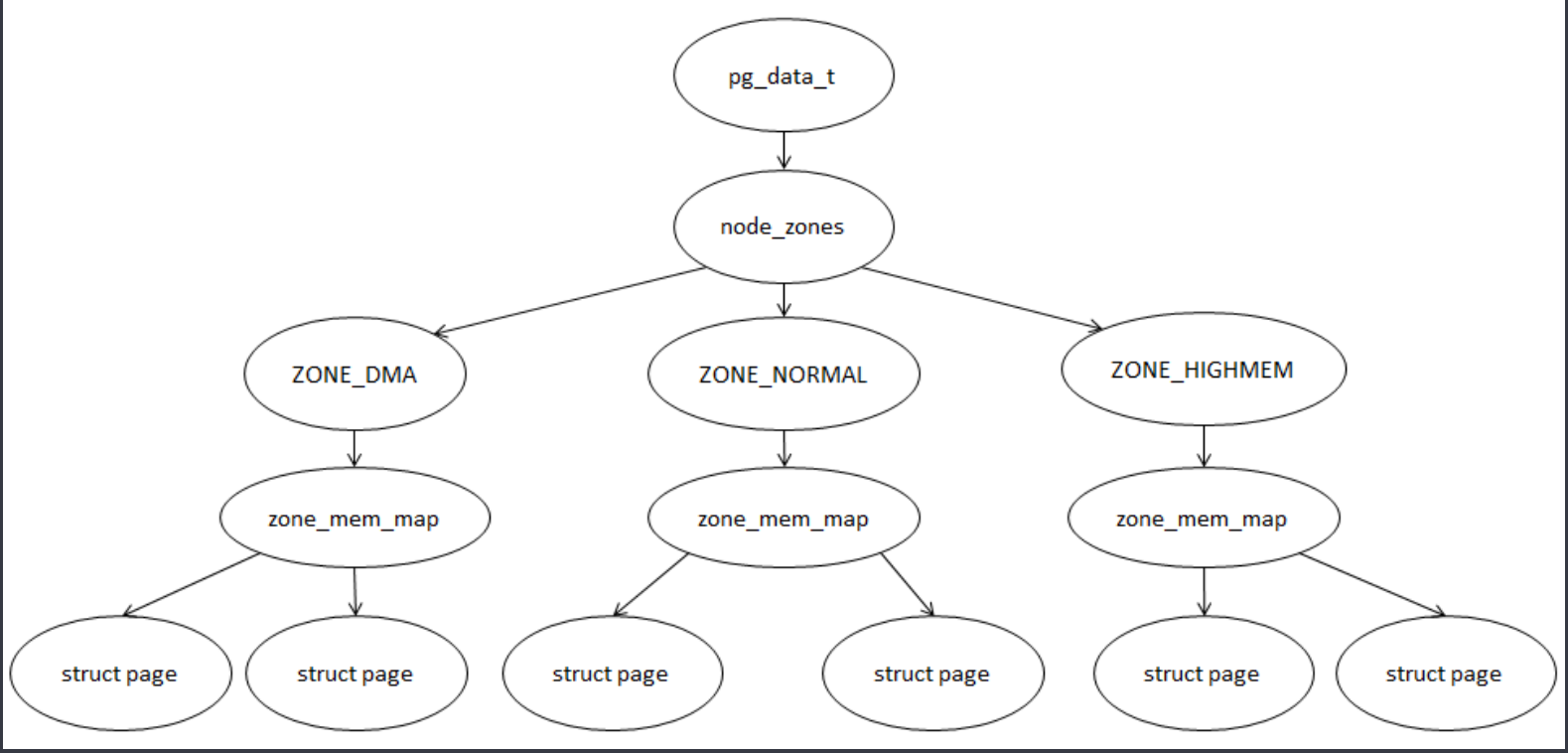
```
enum zone_type {
        ZONE_DMA,
        ZONE_DMA32,
        ZONE_NORMAL,
#ifdef CONFIG_HIGHMEM
        ZONE_HIGHMEM,
#endif
        ZONE_MOVABLE,
#ifdef CONFIG_ZONE_DEVICE
        ZONE_DEVICE,
#endif
        __MAX_NR_ZONES
}
```

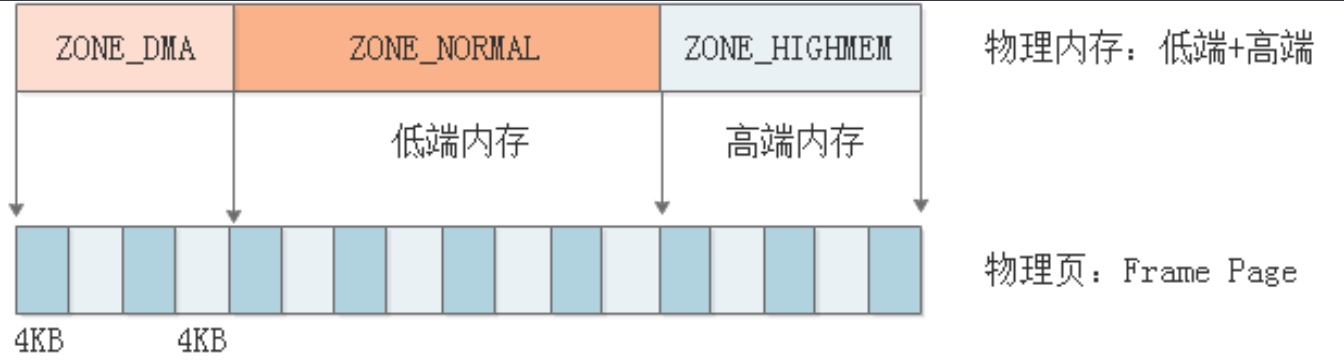
```
struct zone {
        unsigned long           _watermark [NR_WMARK];
        unsigned long           watermark_boost;
        unsigned long           nr_reserved_highatomic;
        long                    lowmem_reserve [MAX_NR_ZONES];
        struct pglist_data      *zone_pgdat;
        struct per_cpu_pageset  __percpu *pageset;
        /* zone_start_pfn == zone_start_paddr >> PAGE_SHIFT */
        unsigned long           zone_start_pfn;
        atomic_long_t           managed_pages;
        unsigned long           spanned_pages;
        unsigned long           present_pages;
        const char              *name;
        int initialized;
        struct free_area        free_area[MAX_ORDER];
        unsigned long           flags;
        unsigned long           percpu_drift_mark;
#if defined CONFIG_COMPACTION || defined CONFIG_CMA
        unsigned long           compact_cached_free_pfn;
        unsigned long           compact_cached_migrate_pfn[ASYNC_AND_SYNC];
        unsigned long           compact_init_migrate_pfn;
        unsigned long           compact_init_free_pfn;
#endif
        bool                    contiguous;
        atomic_long_t           vm_stat[NR_VM_ZONE_STAT_ITEMS];
        atomic_long_t           vm_numa_stat[NR_VM_NUMA_STAT_ITEMS];
}
```

- 内存节点：node
  - 内存模型：UMA和NUMA
  - struct pglist_data：表示node中的内存资源
  - 定义：include/linux/mmzone.h
  - 结构体： struct pglist_data 核心成员解析
  - node_data数组：保存所有node的pglist_data结构

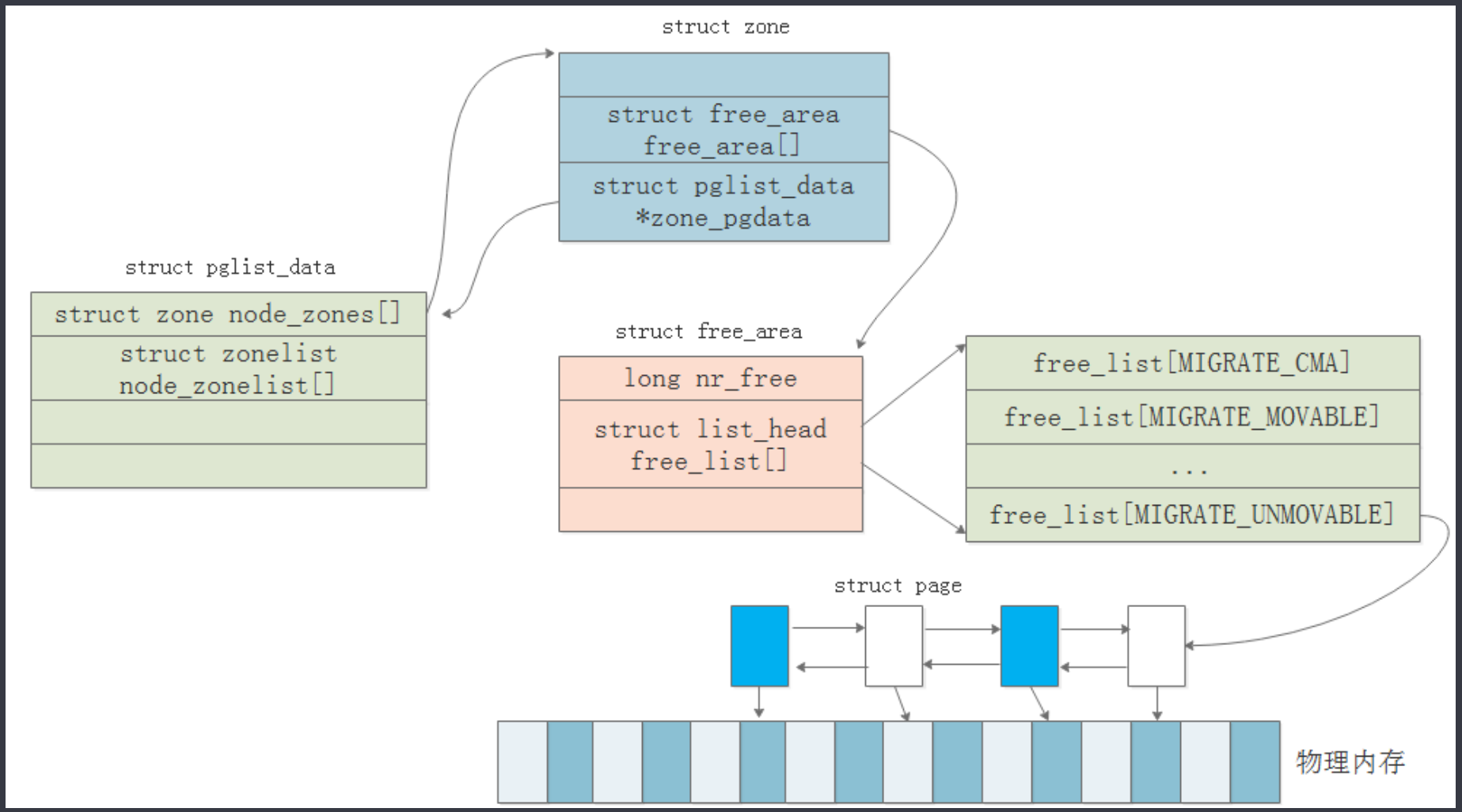
```
typedef struct pglist_data {
        struct zone node_zones[MAX_NR_ZONES];
        unsigned long node_start_pfn;
        unsigned long node_present_pages;     /* total number of physical pages */
        unsigned long node_spanned_pages;    /* total size of physical page*/
        struct page *node_mem_map;
};
```

• 物理内存管理架构


pg_data_t
node_zones
ZONE_DMA
ZONE_NORMAL
ZONE_HIGHMEM
zone_mem_map
zone_mem_map
zone_mem_map
struct page
struct page
struct page
struct page
struct page
struct page

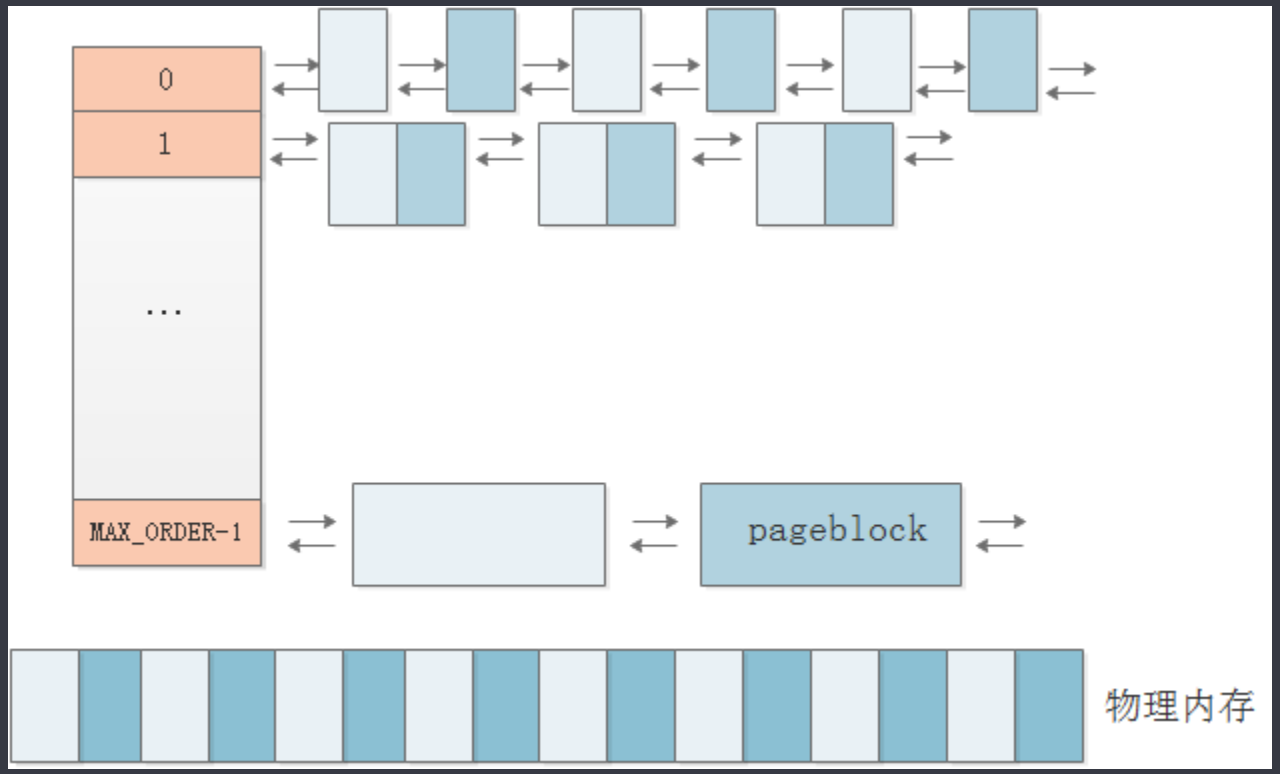

ZONE_DMA
ZONE_NORMAL
ZONE_HIGHMEM
物理内存：低端+高端
低端内存
高端内存
物理页：Frame Page
4KB
4KB

• 核心结构体关联



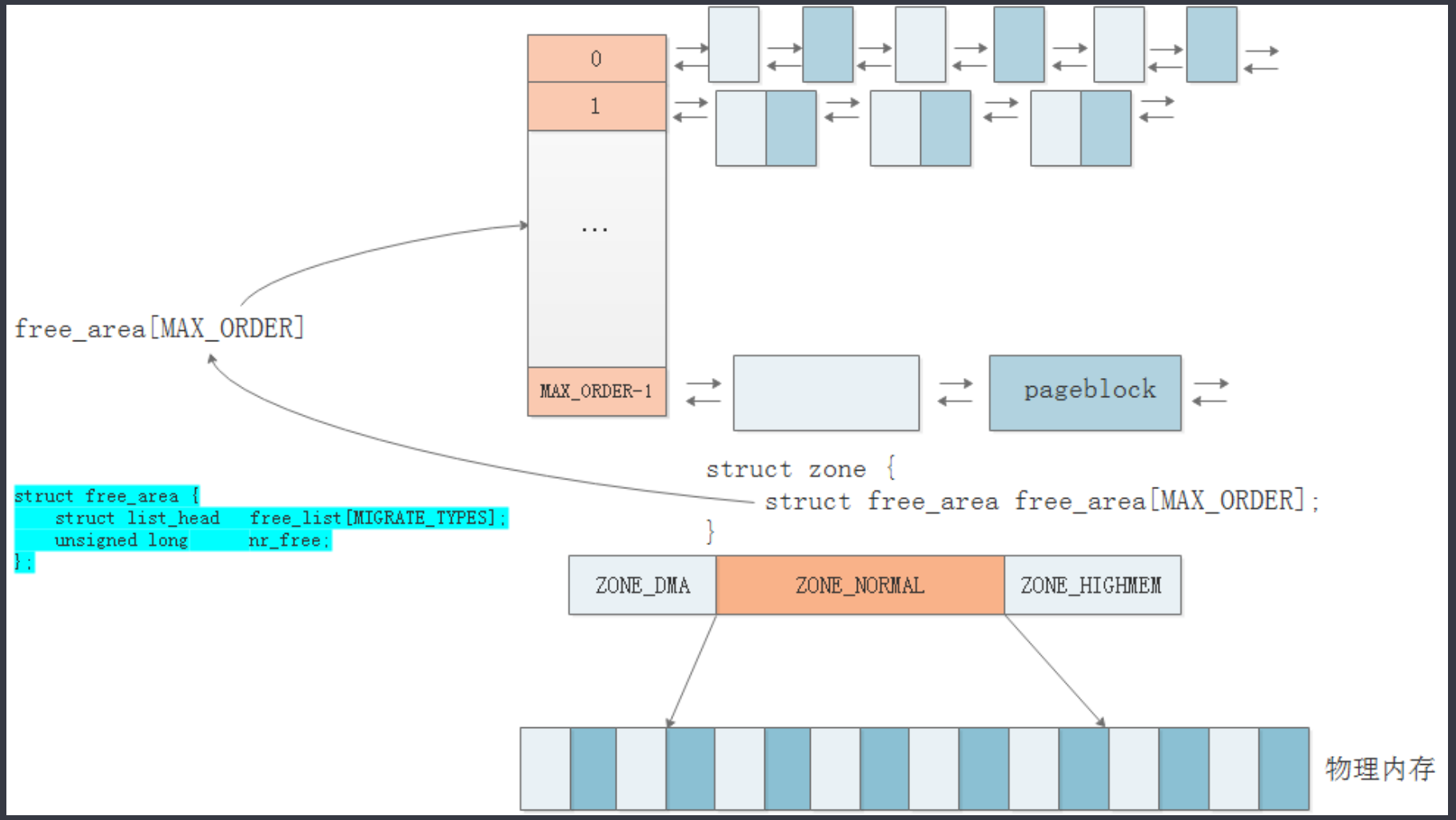
04 伙伴系统：buddy system

- 伙伴系统(buddy system)
  - 物理内存由页分配器(page allocator)接管
  - 内存块的申请、释放过程
  - 伙伴算法、阶数

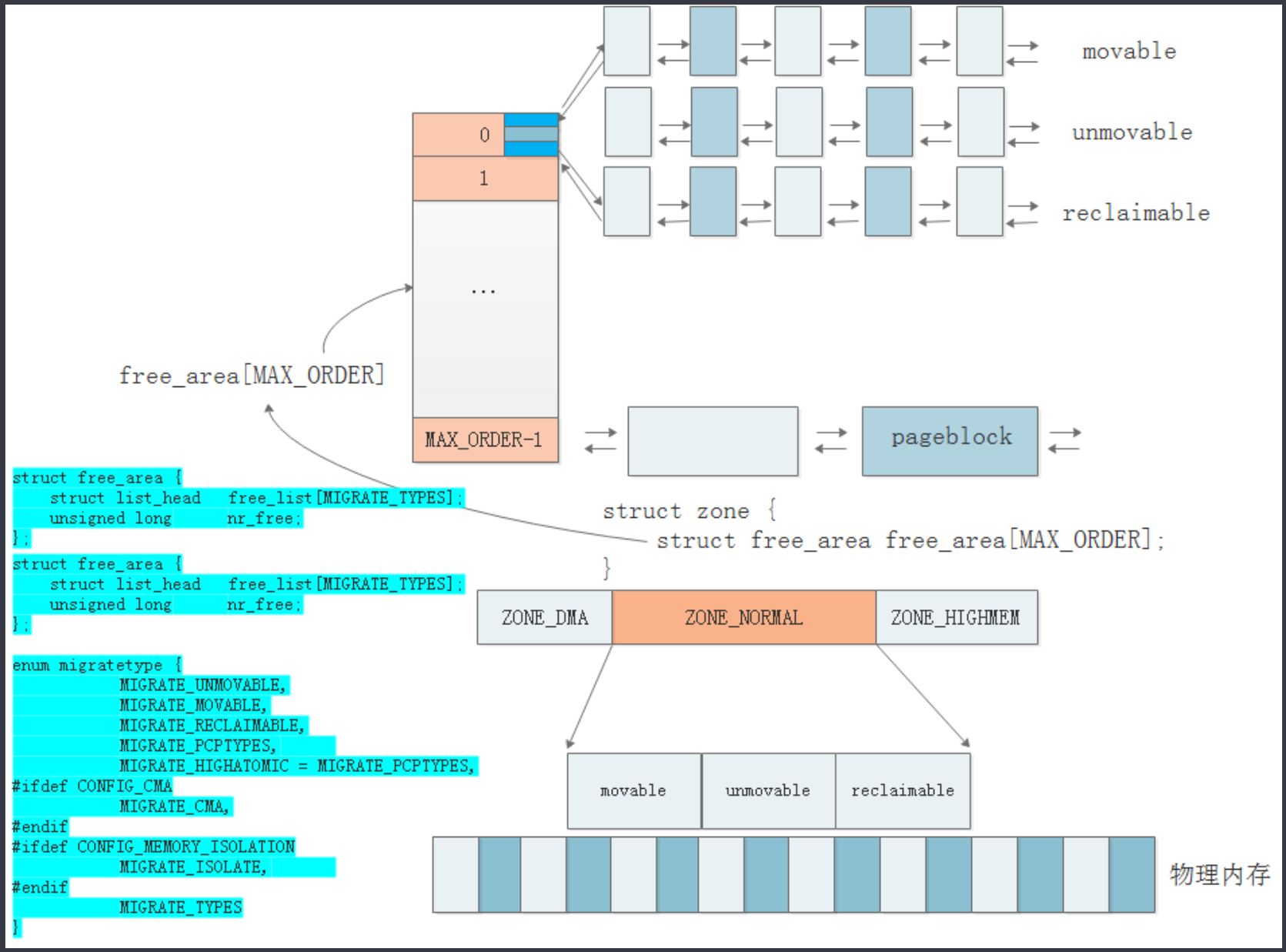


• 伙伴系统(buddy system)

• 核心结构体关联



• 伙伴系统(buddy system)



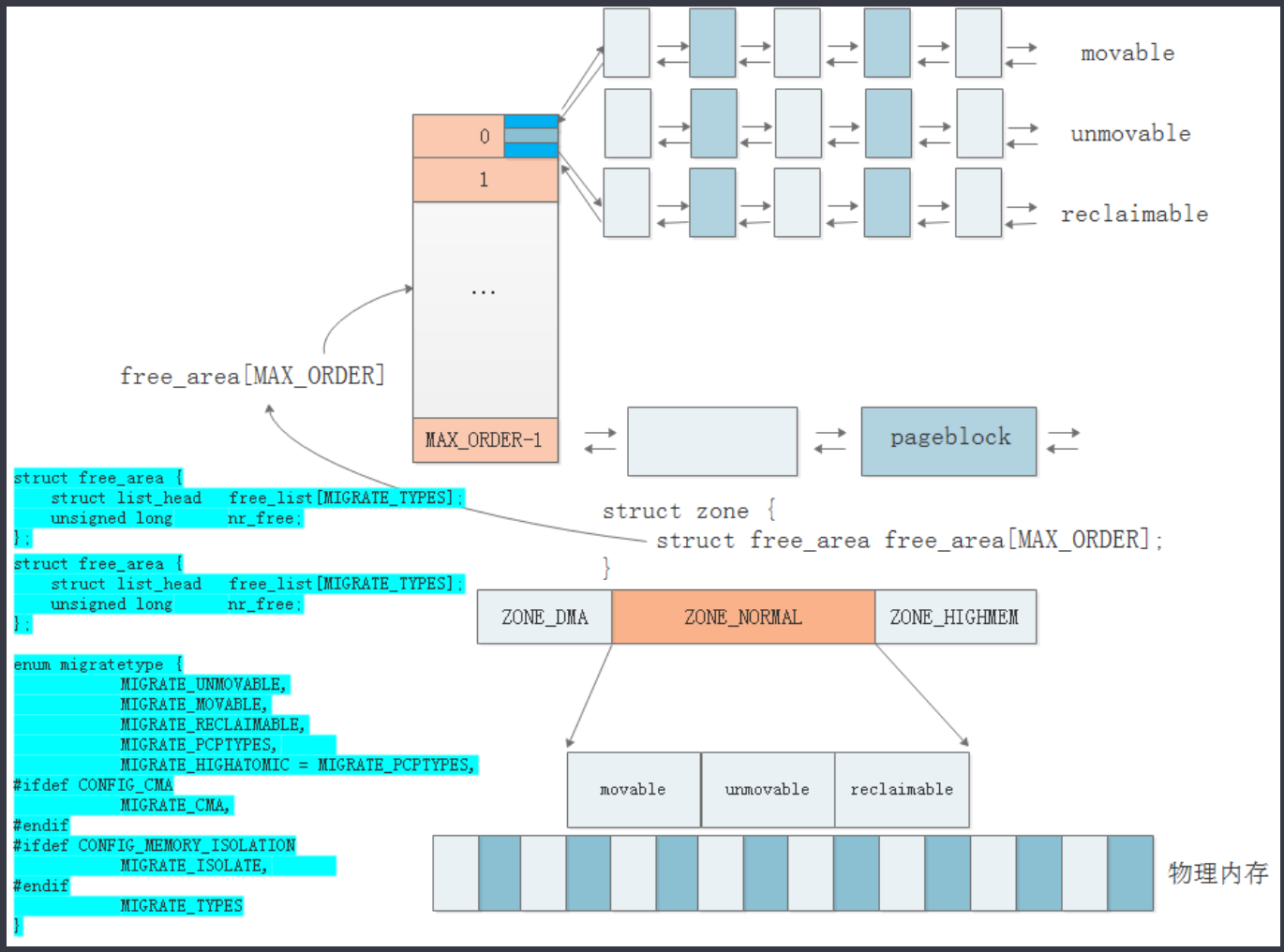
- 伙伴系统(buddy system)
  - 核心结构体关联

```
struct page {
        unsigned long  private; //page_order=1
        atomic_t   _mapcount;
        atomic_t   _refcount;
};
```

05 物理页面的迁移类型：migratetype

• 核心数据结构： struct free_area

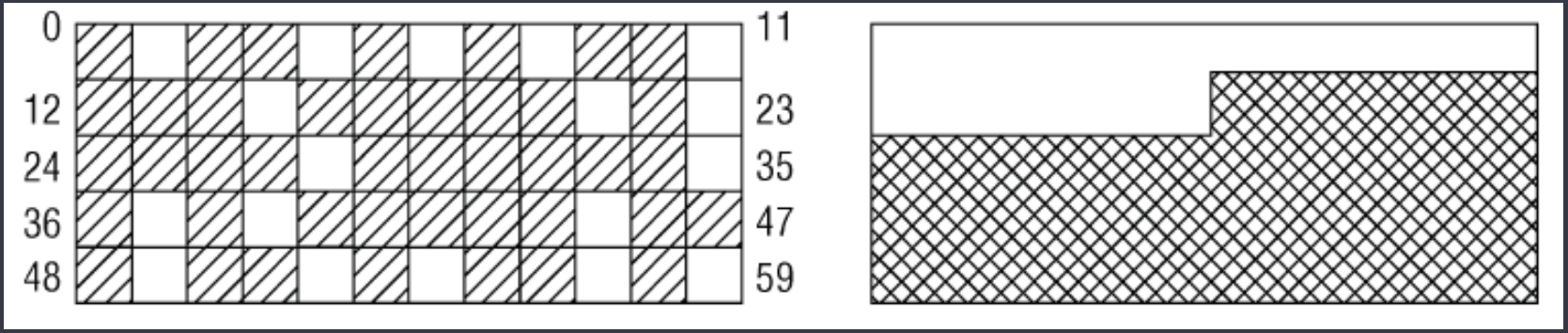
```
struct free_area {
        struct list_head        free_list[MIGRATE_TYPES];
        unsigned long           nr_free;
};
```



- 枚举类型： migratetype
  - 查看页面迁移类型： # cat /proc/pagetypeinfo
    - 可移动的：用户进程申请的内存
    - 可回收的：文件系统的page cache
    - 不可移动的：内核镜像区的物理内存

```
enum migratetype {
        MIGRATE_UNMOVABLE,
        MIGRATE_MOVABLE,
        MIGRATE_RECLAIMABLE,
        MIGRATE_PCPTYPES,       /* the number of types on the pcp lists */
        MIGRATE_HIGHATOMIC = MIGRATE_PCPTYPES,
#ifdef CONFIG_CMA
        MIGRATE_CMA,
#endif
#ifdef CONFIG_MEMORY_ISOLATION
        MIGRATE_ISOLATE, /* can't allocate from here */
#endif
        MIGRATE_TYPES
};
```

- 为什么要引入迁移类型？
  - 伙伴系统存在的问题
  - 对伙伴系统的改进
  - page migratetype：不同类型的页面分类存储
  - memory compaction： 内存碎片整理



- 内存碎片整理：memory compaction
  - 碎片清理：可移动页面迁移
  - memory compact的几种方式
    - COMPACT_PRIO_SYNC_FULL：以同步方式压缩和迁移
    - COMPACT_PRIO_SYNC_LIGHT：压缩同步，迁移异步
    - COMPACT_PRIO_ASYNC：以异步方式压缩和迁移
  - 什么时候会触发memory compaction？
    - kcompacted守护线程
    - memory compaction开销

06 Per-CPU页帧缓存

- Per-CPU页帧缓存
  - Per-CPU同步机制
  - pcp： per CPU pages
  - 单个物理页帧的申请与释放

```
struct zone {
        unsigned long           _watermark [NR_WMARK];
        unsigned long           watermark_boost;
        unsigned long           nr_reserved_highatomic;
        long                    lowmem_reserve [MAX_NR_ZONES];
        struct pglist_data      *zone_pgdat;
        struct per_cpu_pageset   __percpu *pageset;
        /* zone_start_pfn == zone_start_paddr >> PAGE_SHIFT */
        unsigned long           zone_start_pfn;
        unsigned long           present_pages;
        const char              *name;
        int initialized;
        struct free_area        free_area[MAX_ORDER];
        unsigned long           flags;
        unsigned long           percpu_drift_mark;
}
```

• 核心结构体

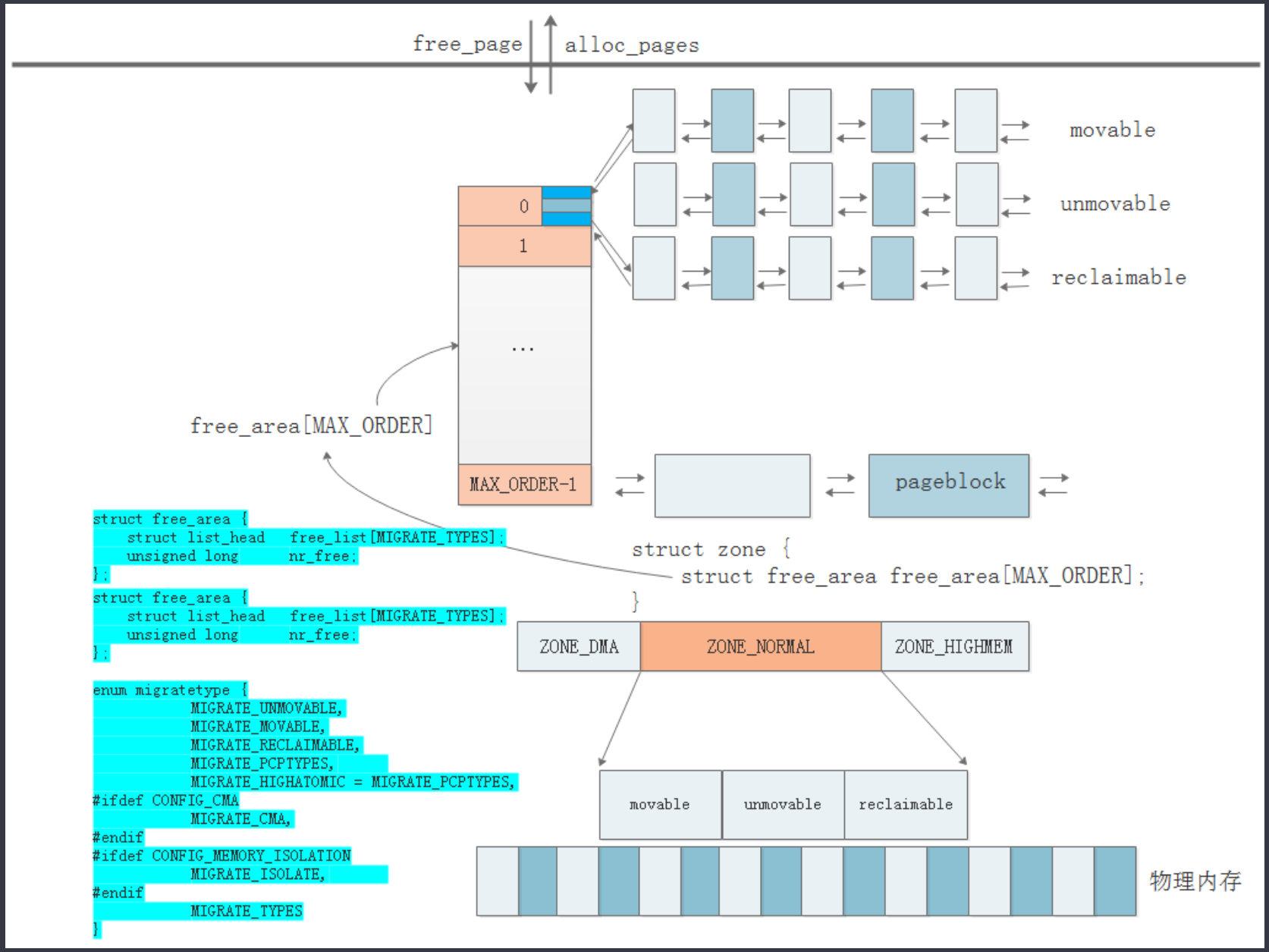
```
struct per_cpu_pages {
        int count;          /* number of pages in the list */
        int high;           /* high watermark, emptying needed */
        int batch;          /* chunk size for buddy add/remove */
        /* Lists of pages, one per migrate type stored on the pcp-lists */
        struct list_head  lists[MIGRATE_PCPTYPES];
};

struct per_cpu_pageset {
        struct per_cpu_pages  pcp;
        …
};

Struct zone {
         struct per_cpu_pageset __percpu *pageset;
         struct free_area    free_area[MAX_ORDER];
}
```

07 页分配器接口：alloc_pages

• 伙伴分配器接口


free_page
alloc_pages
movable
unmovable
reclaimable
0
1
...
MAX_ORDER-1
free_area[MAX_ORDER]
pageblock
struct free_area {
struct list_head free_list[MIGRATE_TYPES];
unsigned long nr_free;
};
struct free_area {
struct list_head free_list[MIGRATE_TYPES];
unsigned long nr_free;
};
struct zone {
struct free_area free_area[MAX_ORDER];
}
ZONE_DMA
ZONE_NORMAL
ZONE_HIGHMEM
enum migratetype {
MIGRATE_UNMOVABLE,
MIGRATE_MOVABLE,
MIGRATE_RECLAIMABLE,
MIGRATE_PCPTYPES,
MIGRATE_HIGHATOMIC = MIGRATE_PCPTYPES,
#ifdef CONFIG_CMA
MIGRATE_CMA,
#endif
#ifdef CONFIG_MEMORY_ISOLATION
MIGRATE_ISOLATE,
#endif
MIGRATE_TYPES
}
movable
unmovable
reclaimable
物理内存

- 页分配器接口
  - 头文件：include/linux/gfp.h
  - 编程示例：使用页分配器接口申请内存
  - 内核源码分析

```
struct page *alloc_pages(gfp_t gfp_mask, unsigned int order);
#define alloc_page(gfp_mask) alloc_pages(gfp_mask, 0)
unsigned long __get_free_pages(gfp_t gfp_mask, unsigned int order);
#define __get_free_page(gfp_mask)  __get_free_pages((gfp_mask), 0)

void free_pages(unsigned long addr, unsigned int order);
void __free_pages(struct page *page, unsigned int order);
#define __free_page(page) __free_pages((page), 0)
#define free_page(addr) free_pages((addr), 0)
```

• 掩码gfp_mask

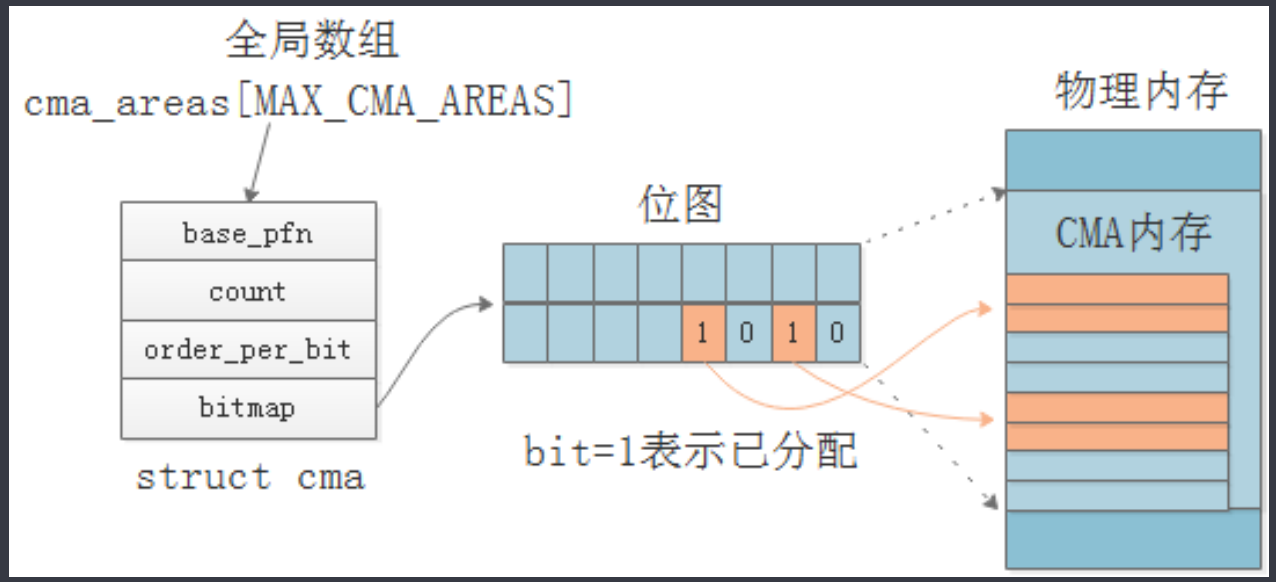
```
#define GFP_ATOMIC      (__GFP_HIGH|__GFP_ATOMIC|__GFP_KSWAPD_RECLAIM)
#define GFP_KERNEL      (__GFP_RECLAIM | __GFP_IO | __GFP_FS)
#define GFP_KERNEL_ACCOUNT (GFP_KERNEL | __GFP_ACCOUNT)
#define GFP_NOWAIT      (__GFP_KSWAPD_RECLAIM)
#define GFP_NOIO        (__GFP_RECLAIM)
#define GFP_NOFS        (__GFP_RECLAIM | __GFP_IO)
#define GFP_USER        (__GFP_RECLAIM | __GFP_IO | __GFP_FS | __GFP_HARDWALL)
#define GFP_DMA         __GFP_DMA
#define GFP_DMA32       __GFP_DMA32
#define GFP_HIGHUSER    (GFP_USER | __GFP_HIGHMEM)
#define GFP_HIGHUSER_MOVABLE    (GFP_HIGHUSER | __GFP_MOVABLE)
#define GFP_TRANSHUGE_LIGHT     ((GFP_HIGHUSER_MOVABLE | __GFP_COMP | \
                                  __GFP_NOMEMALLOC | __GFP_NOWARN) & ~__GFP_RECLAIM)
#define GFP_TRANSHUGE           (GFP_TRANSHUGE_LIGHT | __GFP_DIRECT_RECLAIM)

/* Convert GFP flags to their corresponding migrate type */
#define GFP_MOVABLE_MASK (__GFP_RECLAIMABLE|__GFP_MOVABLE)
```

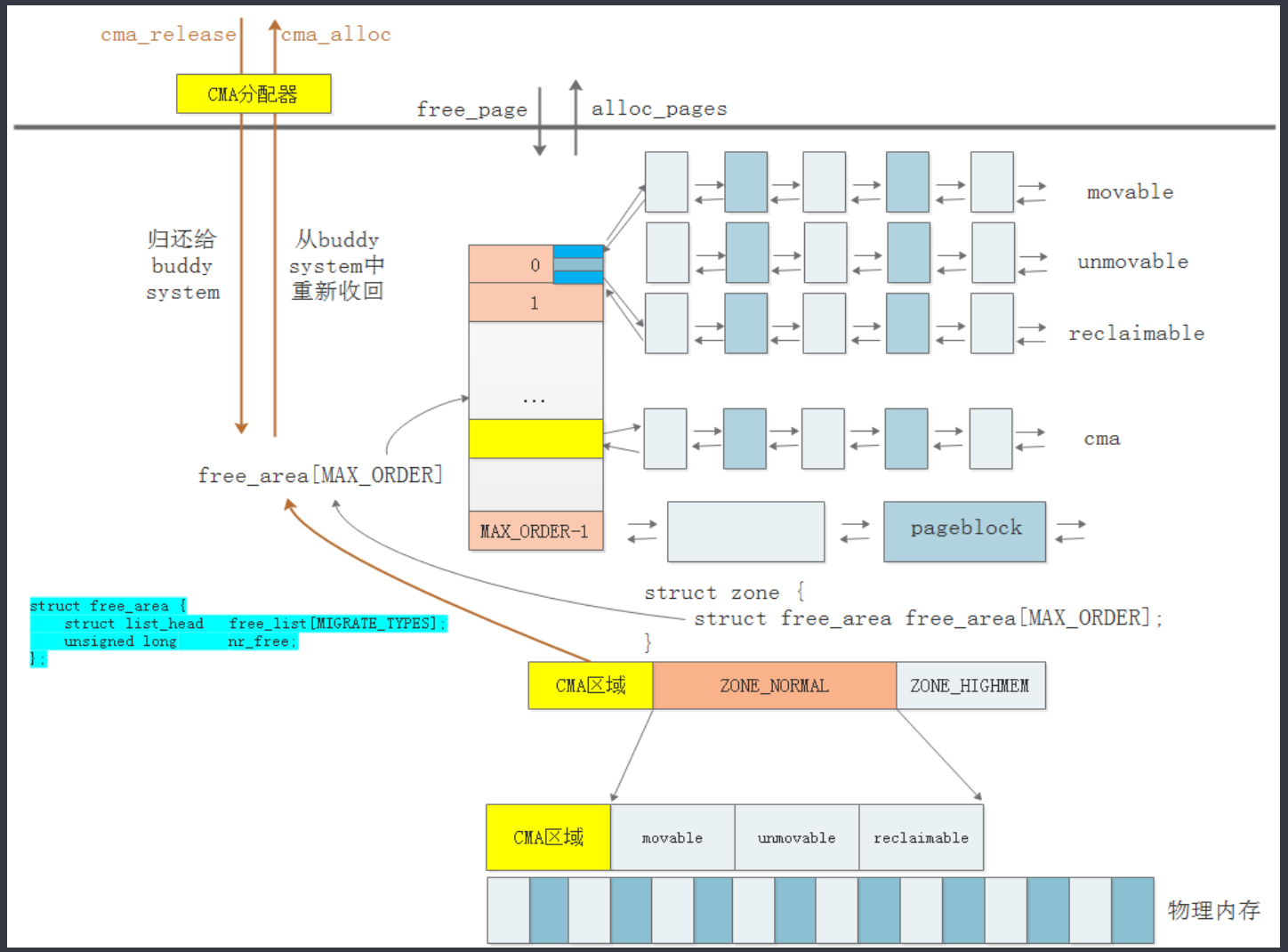
08 连续内存分配器：CMA

- CMA： Contiguous Memory Allocator
  - 如何申请一大块连续的物理内存？
  - CMA初始化： 内核配置、设备树dts文件

```
struct cma {
                    unsigned long   base_pfn;
                    unsigned long   count;
                    unsigned long   *bitmap;
                    unsigned int order_per_bit;  //=11
                    struct mutex    lock;
                    char name[CMA_MAX_NAME];
};
extern struct  cma  cma_areas[MAX_CMA_AREAS];
#define  MAX_CMA_AREAS          (1 + CONFIG_CMA_AREAS)
extern unsigned cma_area_count;
```

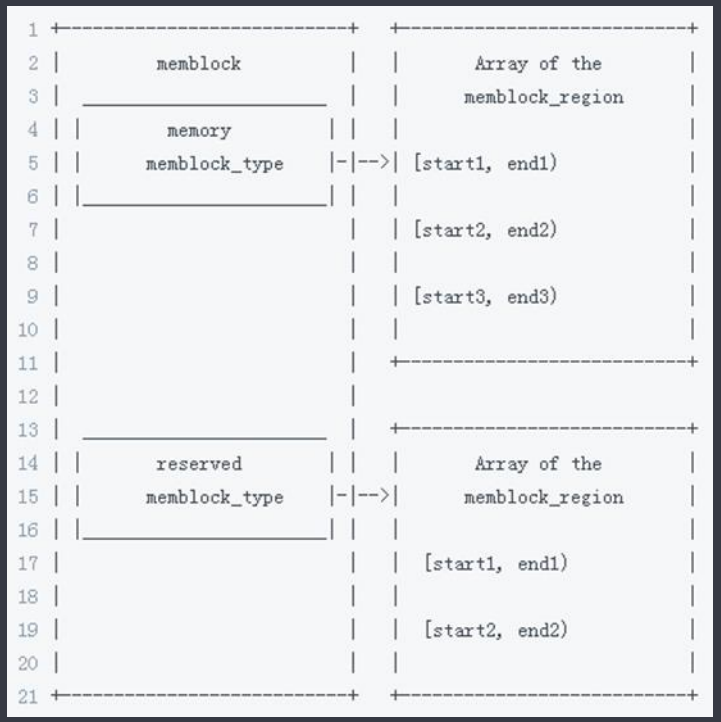


- CMA接口： cma_alloc、 cma_release
  - MIGRATE_ISOLATE、MIGRATE_CMA



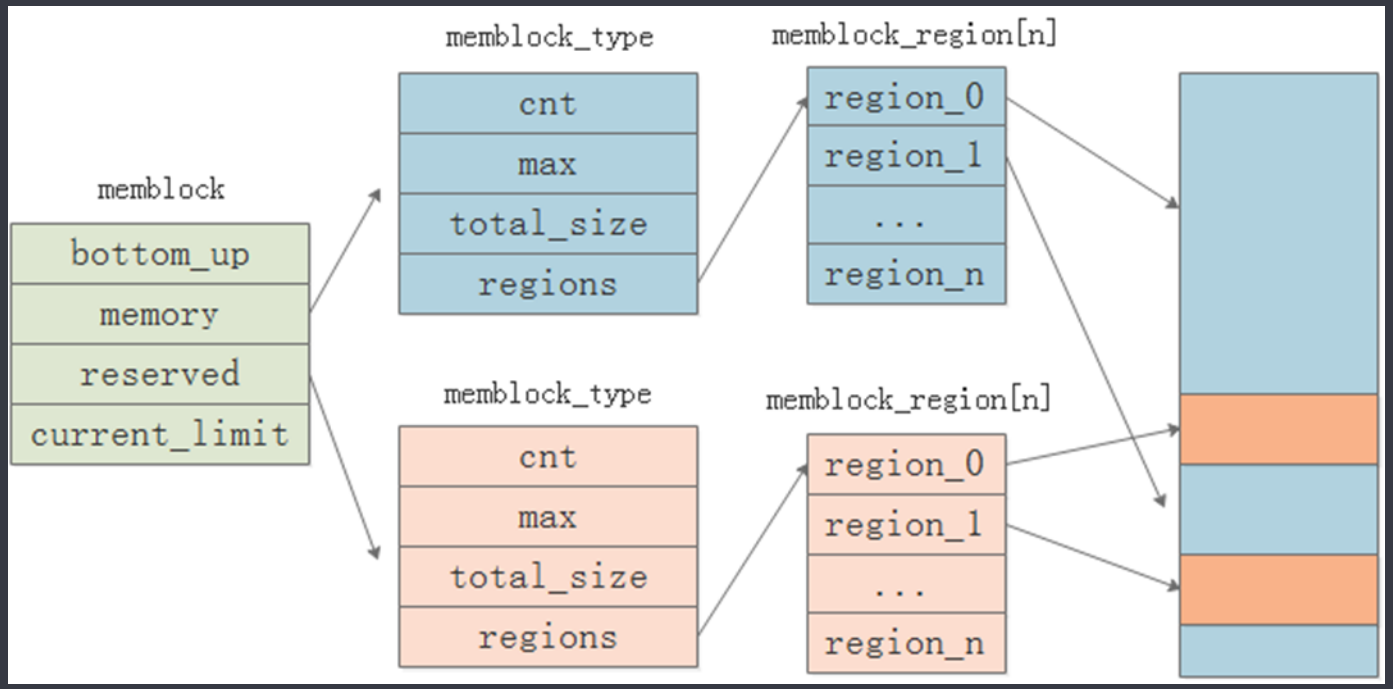
09 伙伴系统初始化(一)：memblock管理器

- 早期的内存管理
  - 内核如何获取内存的地址、大小
  - 全局变量： struct memblock memblock;
    - 可用的物理内存： memblock.memory 数组
    - Reserve的物理内存： memblock.reserved 数组
      - » 内核镜像(.init段除外)、dtb、U-boot、页表
      - » GPU、camera、 音视频编解码



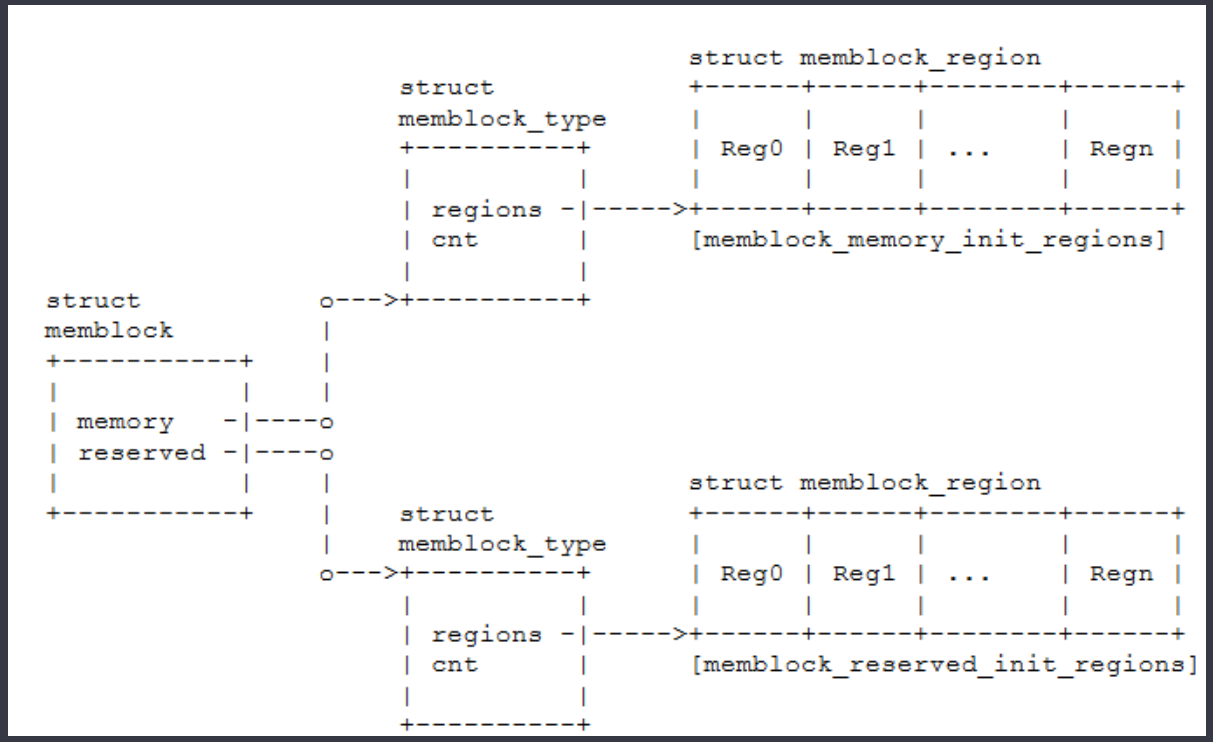
- memblock 接口
  - /sys/kernel/debug/memblock/memory
  - /sys/kernel/debug/memblock/reserved
  - /proc/kmsg： memblock=debug

```
int memblock_add (phys_addr_t base, phys_addr_t size);
int memblock_remove (phys_addr_t base, phys_addr_t size);
for_each_mem_range
int memblock_reserve (phys_addr_t base, phys_addr_t size);
int memblock_free (phys_addr_t base, phys_addr_t size);
```



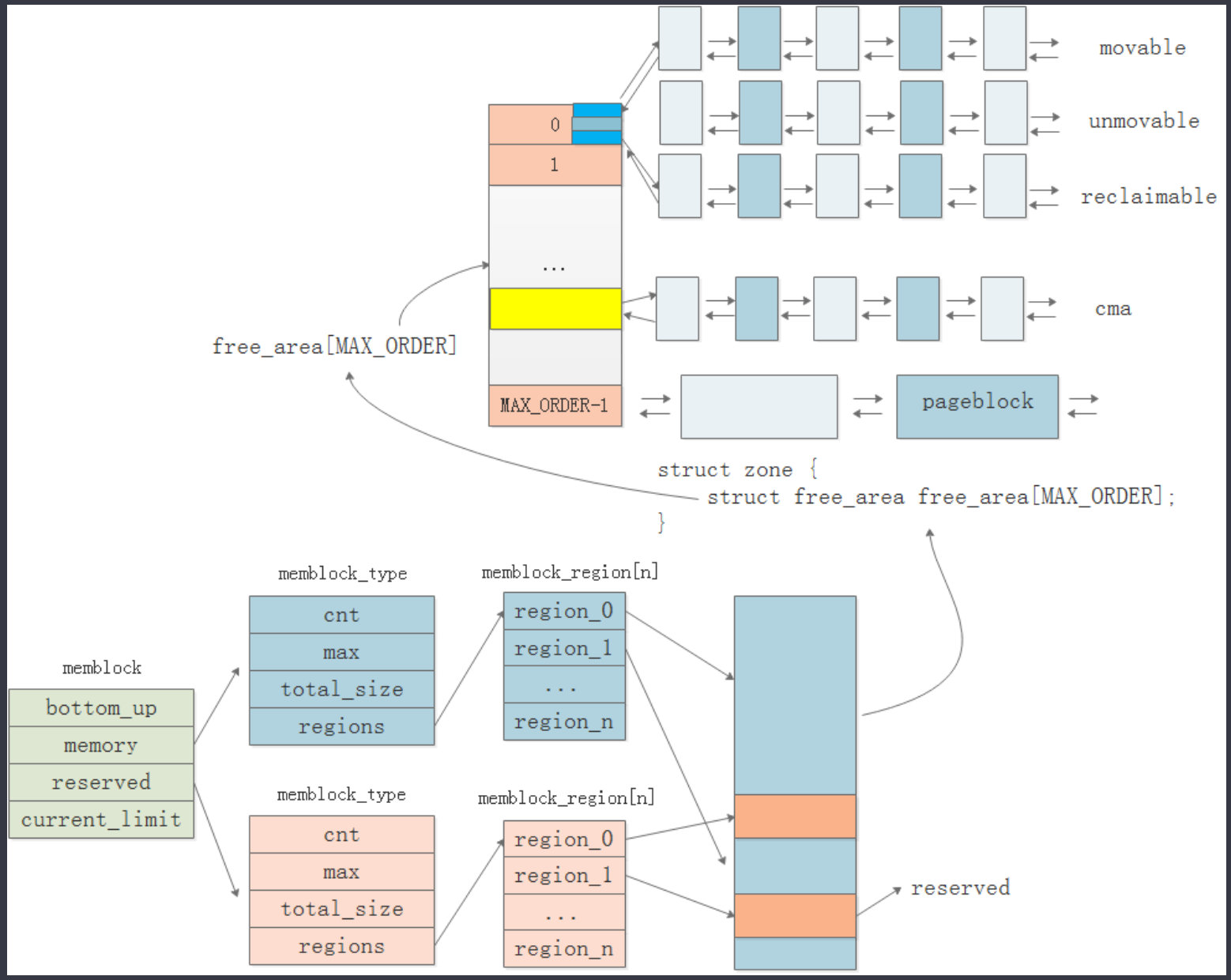
- memblock的初始化
  - 获取物理内存的起始地址和大小
  - 初始化全局变量memblock的两个数组

```
setup_arch()->setup_machine_fdt()->early_init_dt_scan()
                                    ->early_init_dt_scan_memory()
        -> arm_memblock_init -> early_init_fdt_scan_reserved_mem
memblock_memory_init_regions[INIT_MEMBLOCK_REGIONS]
memblock_reserved_init_regions[INIT_MEMBLOCK_RESERVED_REGIONS]
```



10 伙伴系统初始化(二)：memblock内存释放

- memblock如何释放内存给伙伴系统？



- memblock如何释放内存给伙伴系统？
  - 内核源码分析：memblock_free_all

```
start_kernel -> setup_arch()->setup_machine_fdt()->early_init_dt_scan()
                                                  ->early_init_dt_scan_memory()
             -> arm_memblock_init -> early_init_fdt_scan_reserved_mem
             -> mm_init -> mem_init -> memblock_free_all

#define for_each_reserved_mem_range(i, p_start, p_end)                    \
        __for_each_mem_range(i, &memblock.reserved, NULL,  NUMA_NO_NODE,
        MEMBLOCK_NONE, p_start, p_end, NULL)

#define for_each_free_mem_range(i, nid, flags, p_start, p_end, p_nid)     \
        __for_each_mem_range(i, &memblock.memory, &memblock.reserved,     \
                             nid, flags, p_start, p_end, p_nid)
```

11 伙伴系统初始化(三)：.init内存释放

- memblock中的reserved memory
  - 内核的代码段(.text/.data/.bss)(.init除外)
  - initrd
  - dtb
  - 设备树中的reserved-memory区域(CMA除外)
  - 临时页表
  - reserved memory的初始化

- .init段的内存释放
  - 为什么要释放？
  - 里面包含了哪些内容？
  - 释放到哪里？
  - 函数：free_initmem分析

```
#define __init    __section(".init.text") __cold  __latent_entropy __noinitretpoline
Static  __init  void func();
start_kernel -> setup_arch()->setup_machine_fdt()->early_init_dt_scan() ->early_init_dt_scan_memory()
                -> arm_memblock_init -> early_init_fdt_scan_reserved_mem
                 -> mm_init -> mem_init -> memblock_free_all
                -> arch_call_rest_init -> rest_init -> kernel_init -> free_initmem
```

```
ALSA device list:
  #0: ARM AC'97 Interface PL041 rev0 at 0x10004000, irq 28
VFS: Mounted root (nfs4 filesystem) on device 0:16.
Freeing unused kernel memory: 1024K
Run /linuxrc as init process
random: fast init done

 __ _(_)_ ___   _ ___   ___ ___
|_  / '_ \ / _` | \ \/ / | | |/ _ \ / __/ __|
 / /| | | | (_| | |>  <| |_| |  __/| (_| (__
/___|_| |_|\__,_|_/_/\_\\__,_|\___(_)___\___|

[root@vexpress ]#
```

12 伙伴系统初始化(四)：CMA内存释放

- CMA内存释放
  - CMA内存如何释放到伙伴系统
  - 内核源码分析

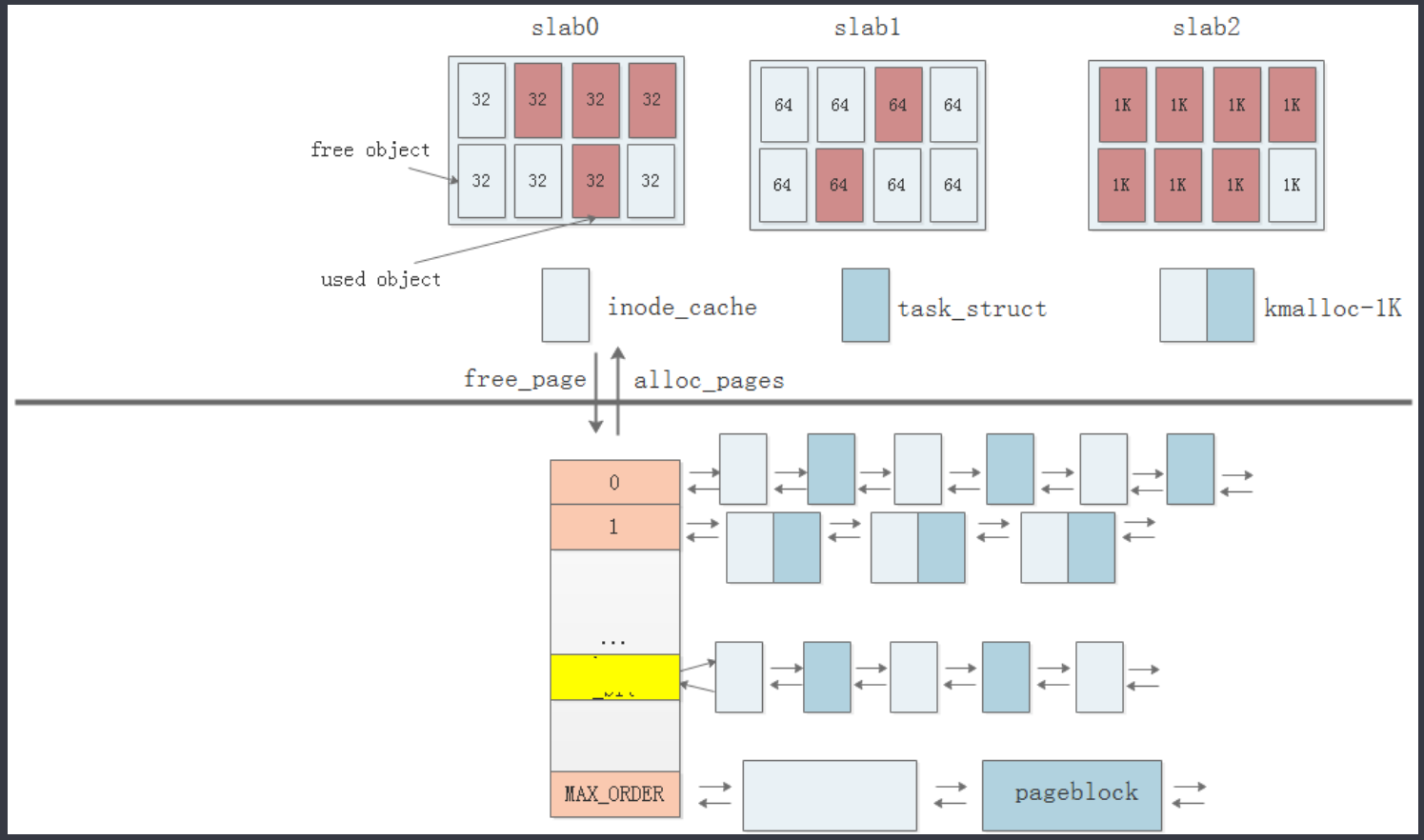
```
reserved_memory.txt
        removed-dma-pool    "linux,dma-default"
        shared-dma-pool     "linux,cma-default"

        no-map
        reusable
```

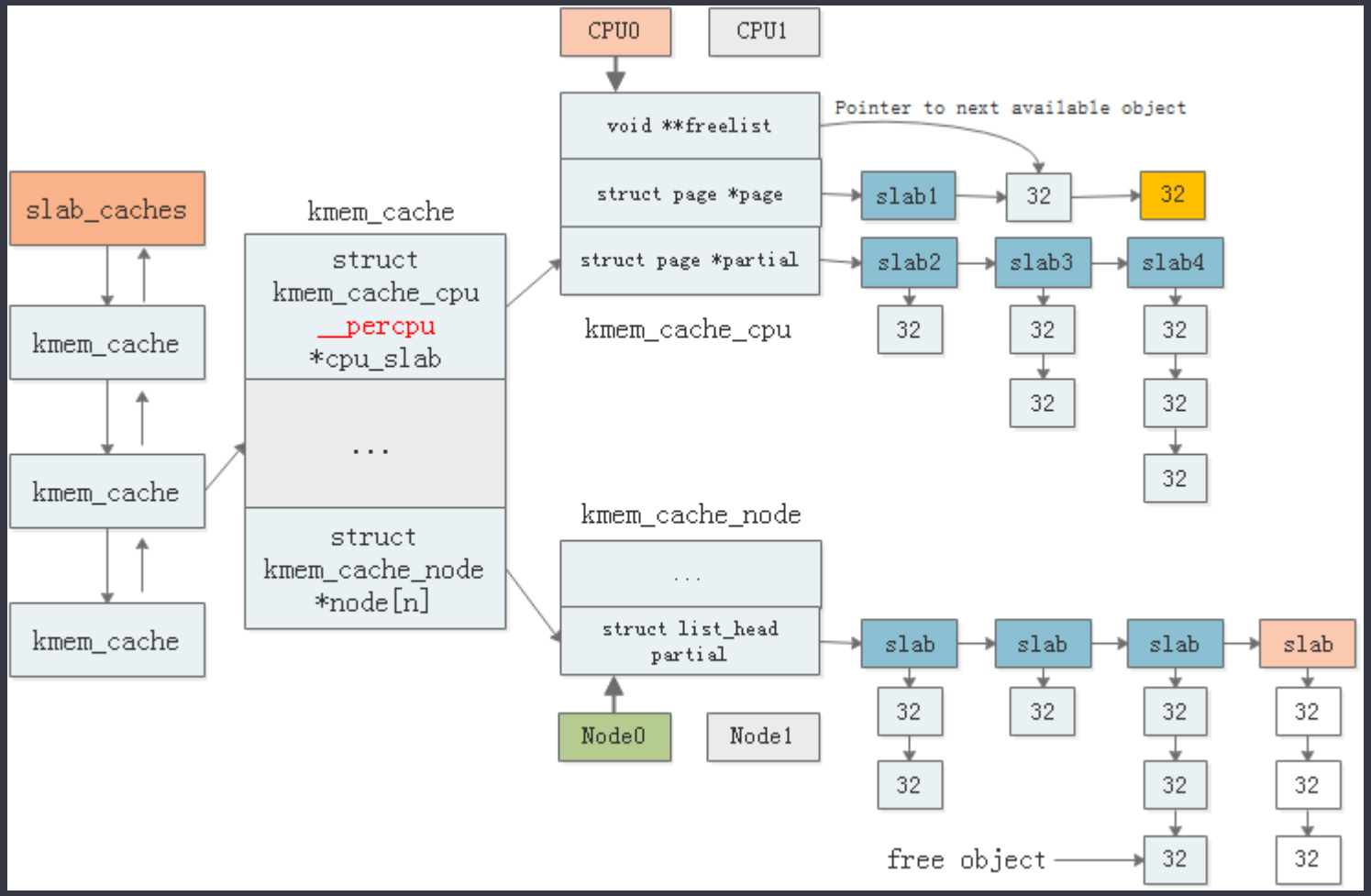
```
start_kernel -> setup_arch()->setup_machine_fdt()->early_init_dt_scan() ->early_init_dt_scan_memory()
        -> arm_memblock_init -> early_init_fdt_scan_reserved_mem
         -> mm_init -> mem_init -> memblock_free_all
         ->core_initcall(cma_init_reserved_areas) -> cma_init_reserved_areas
        -> arch_call_rest_init -> rest_init -> kernel_init -> free_initmem
```

13 slab、slob和slub分配器

- slab工作原理
  - 为什么要引入slab：对伙伴系统的改进和补充
  - slab的工作机制
  - 三种分配器：slab、slob、slub



- slab核心数据结构关联
  - kmem_cache、kmem_cache_cpu、kmem_cache_node
  - 内存的申请和释放分析



- slab编程接口
  - 如何创建一个kmem_cache？
  - 如何创建一个slab？
  - 如何申请一个内存object？
  - 如何释放一个内存object？

```
/*创建和销毁一个kmem_cache*/
struct kmem_cache *kmem_cache_create( const char *name, unsigned int size,
                        unsigned int align, slab_flags_t flags，  void(*ctor)(void*));
struct kmem_cache *kmem_cache_create_usercopy(const char *name,
                                unsigned int size, unsigned int align, slab_flags_t flags,
                                unsigned int useroffset, unsigned int usersize,
                                        void (*ctor)(void *));
int kmem_cache_destroy( struct kmem_cache *cachep);

/*从kmem_cache对应的slab中分配一个object */
void* kmem_cache_alloc(struct kmem_cache* cachep, gfp_t flags);

/*释放object给原先的slab*/
void kmem_cache_free(struct kmem_cache* cachep,  void* objp);
```

- 参数

```
#define SLAB_CONSISTENCY_CHECKS  ((slab_flags_t __force)0x00000100U)
#define SLAB_RED_ZONE            ((slab_flags_t __force)0x00000400U)   /* DEBUG: Red zone objs in a cache */
#define SLAB_POISON              ((slab_flags_t __force)0x00000800U)   /* DEBUG: Poison objects */
#define SLAB_HWCACHE_ALIGN       ((slab_flags_t __force)0x00002000U)  /* Align objs on cache lines */
#define SLAB_CACHE_DMA           ((slab_flags_t __force)0x00004000U)  /* Use GFP_DMA memory */
#define SLAB_CACHE_DMA32         ((slab_flags_t __force)0x00008000U)  /* Use GFP_DMA32 memory */
#define SLAB_STORE_USER  ((slab_flags_t __force)0x00010000U)  /* DEBUG: Store the last owner for bug hunting */
#define SLAB_PANIC               ((slab_flags_t __force)0x00040000U)   /* Panic if kmem_cache_create() fails */
#define SLAB_TYPESAFE_BY_RCU     ((slab_flags_t __force)0x00080000U)
#define SLAB_MEM_SPREAD          ((slab_flags_t __force)0x00100000U)  /* Spread some memory over cpuset */
#define SLAB_TRACE               ((slab_flags_t __force)0x00200000U)          /* Trace allocations and frees */
```

14 kmalloc机制实现分析

- 内存申请编程接口
  - 如何在内核驱动中申请和释放内存？
  - kmalloc实现机制分析
  - kmalloc和slab、伙伴系统的关联

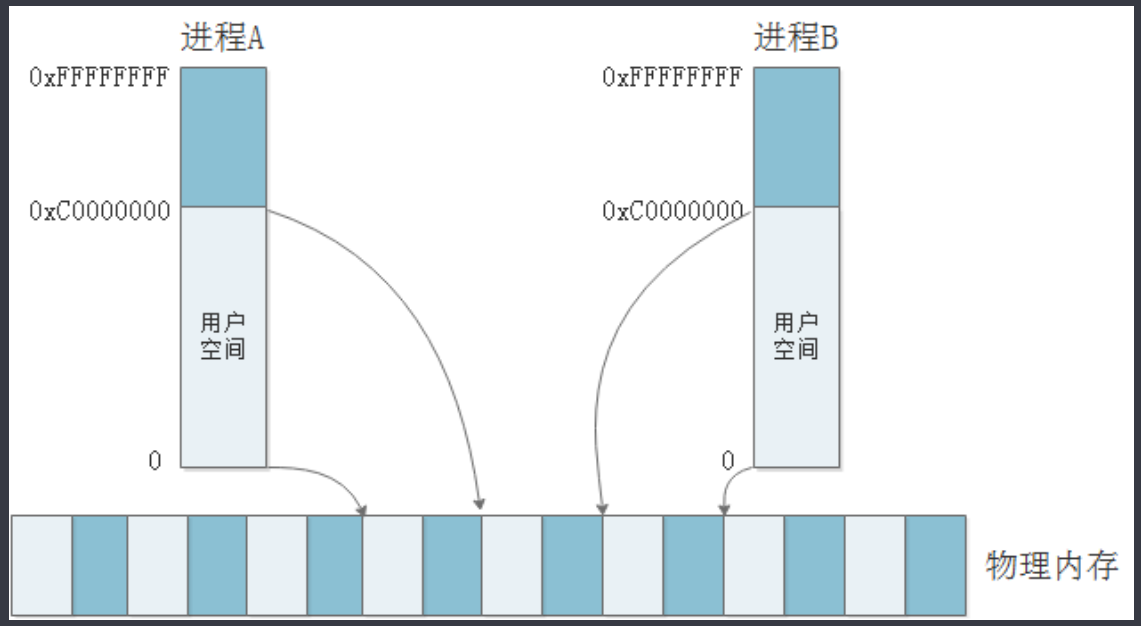
include/linux/slab.h
void *kmalloc(size_t size, gfp_t flags);
void kfree(const void *);

GFP_USER： 由 user 发起的内存申请，可以睡眠
GFP_KERNEL：由内核发起的内存申请，可以睡眠
GFP_ATOMIC：不能睡眠的内存申请请求，比如在中断处理函数中申请内存
GFP_NOWAIT：不等待，不 sleep，申请不到立即返回
GFP_DMA： 从DMA zone分配

- 思考
  - kmalloc能申请的最大内存是多少？
  - Kmalloc返回的地址对齐方式？
  - kmalloc返回的是虚拟地址，还是物理地址？

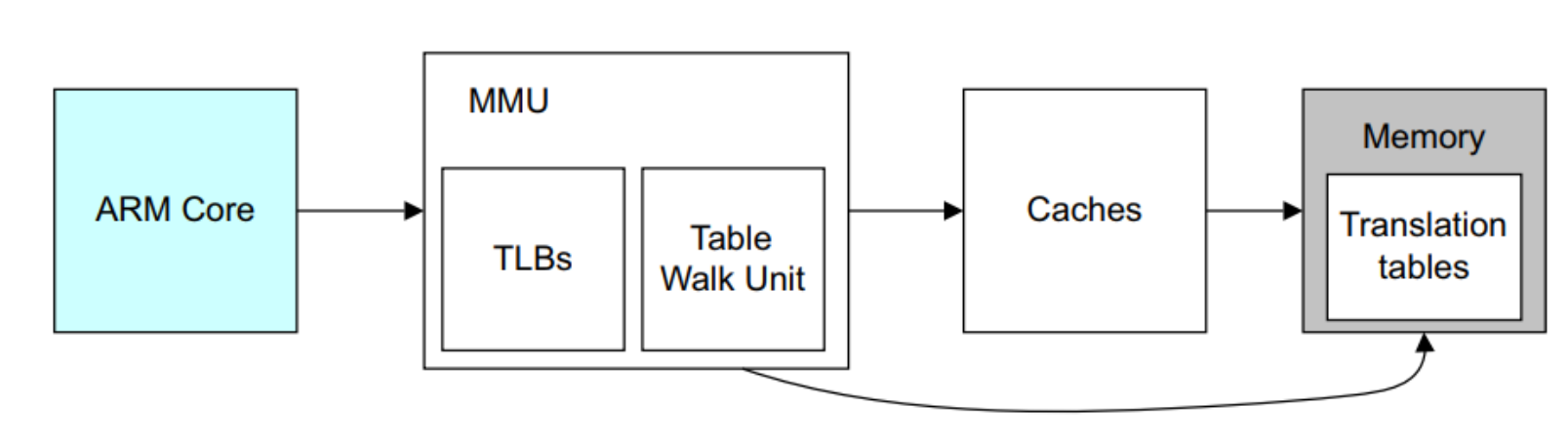
15 虚拟地址和MMU工作原理

- 虚拟地址的基本概念
  - 为什么需要虚拟地址？
  - 虚拟地址、物理地址
  - 线性地址、逻辑地址
  - 总线地址
  - MMU的作用

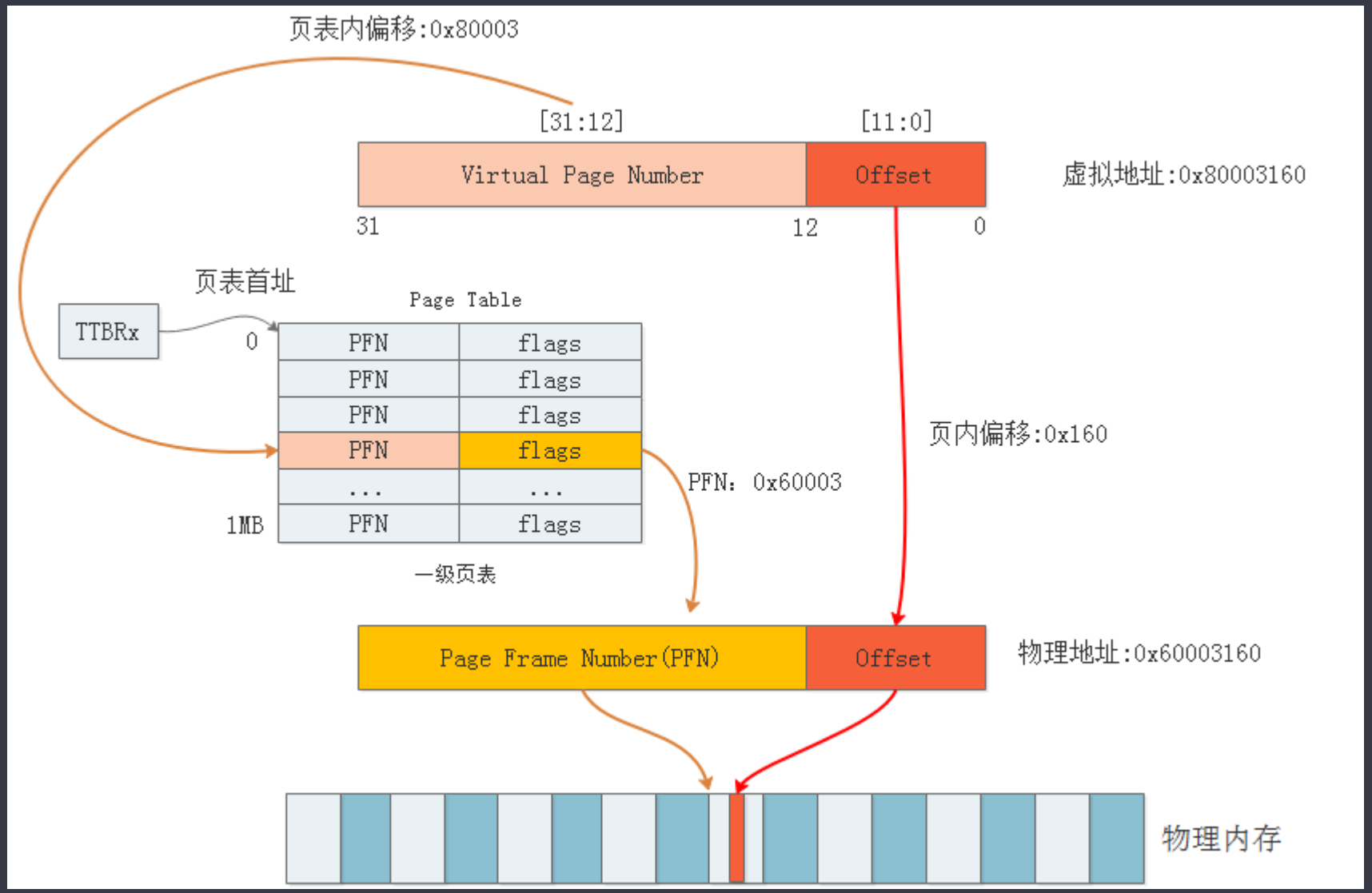


- MMU工作原理
  - 虚拟页号、物理页号
  - 页表
  - 使能MMU
  - TLB、Table Walk Unit

```
MRC p15, 0, R1, c1, C0, 0   ;Read control register
ORR R1, #0x1                ;Set M bit
MCR p15, 0,R1,C1, C0,0      ;Write control register and enable MMU
```

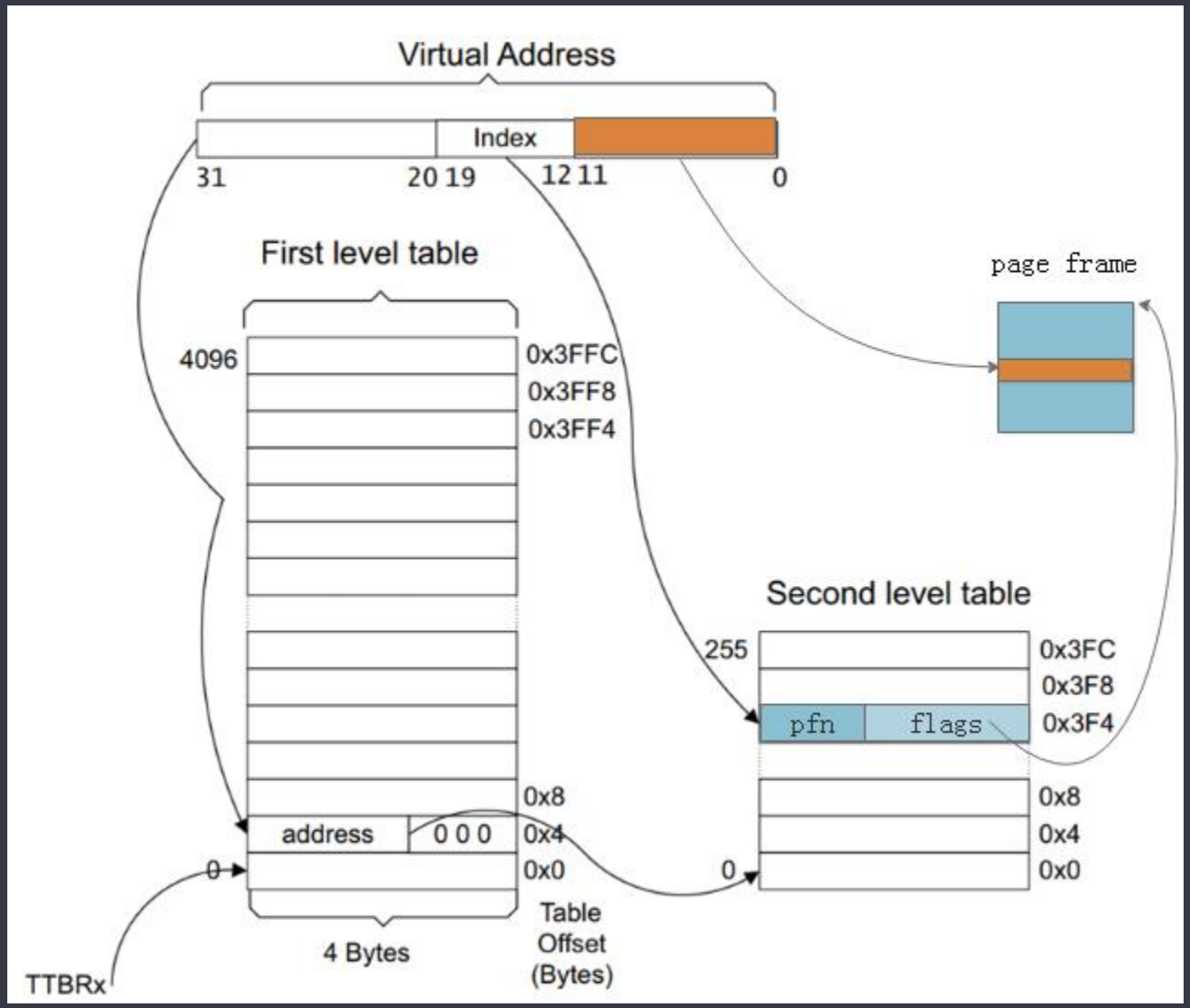


• 虚拟地址到物理地址的转换原理

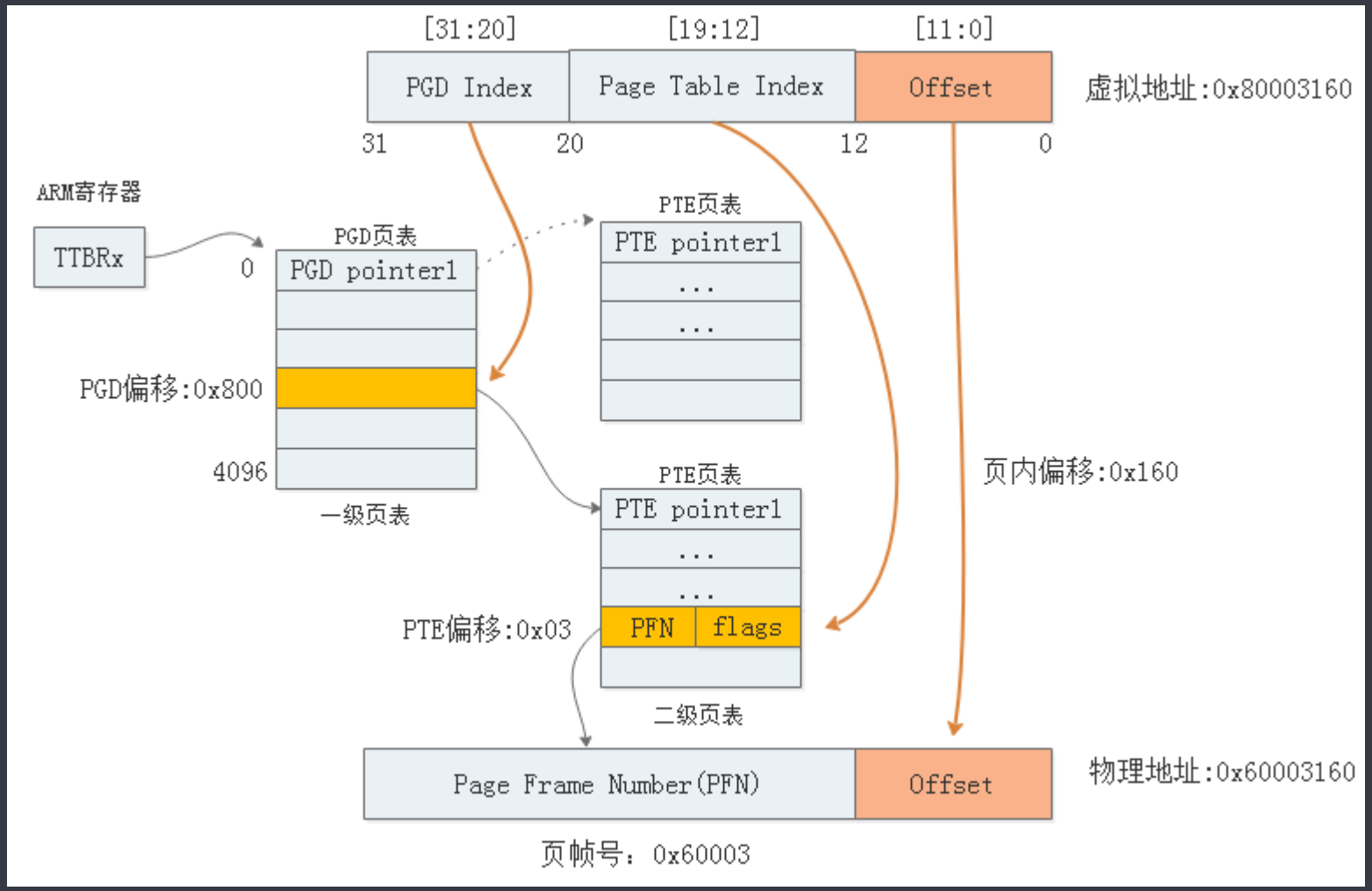


16 二级页表的工作原理

• 二级页表



• 二级页表下的地址转换



- 二级页表的优势
  - 不需要大量连续的物理内存
  - 一个进程不会映射所有的虚拟地址空间
  - 随着页表级数增加，可以节省物理内存

一级页表：

以4KB物理页为映射单位，一个进程4GB的虚拟地址空间需要：

4GB/4KB = 1MB个页表项，每个页表项占用4个字节

每个一级页表需要4MB的存储空间

每个进程需要4MB的内存存储页表，100个进程需要400MB

二级页表：

第一级页表PGD：一共4096项，每个entry4个字节，一共16KB

第二级页表PTE：一共4096个PTE页表

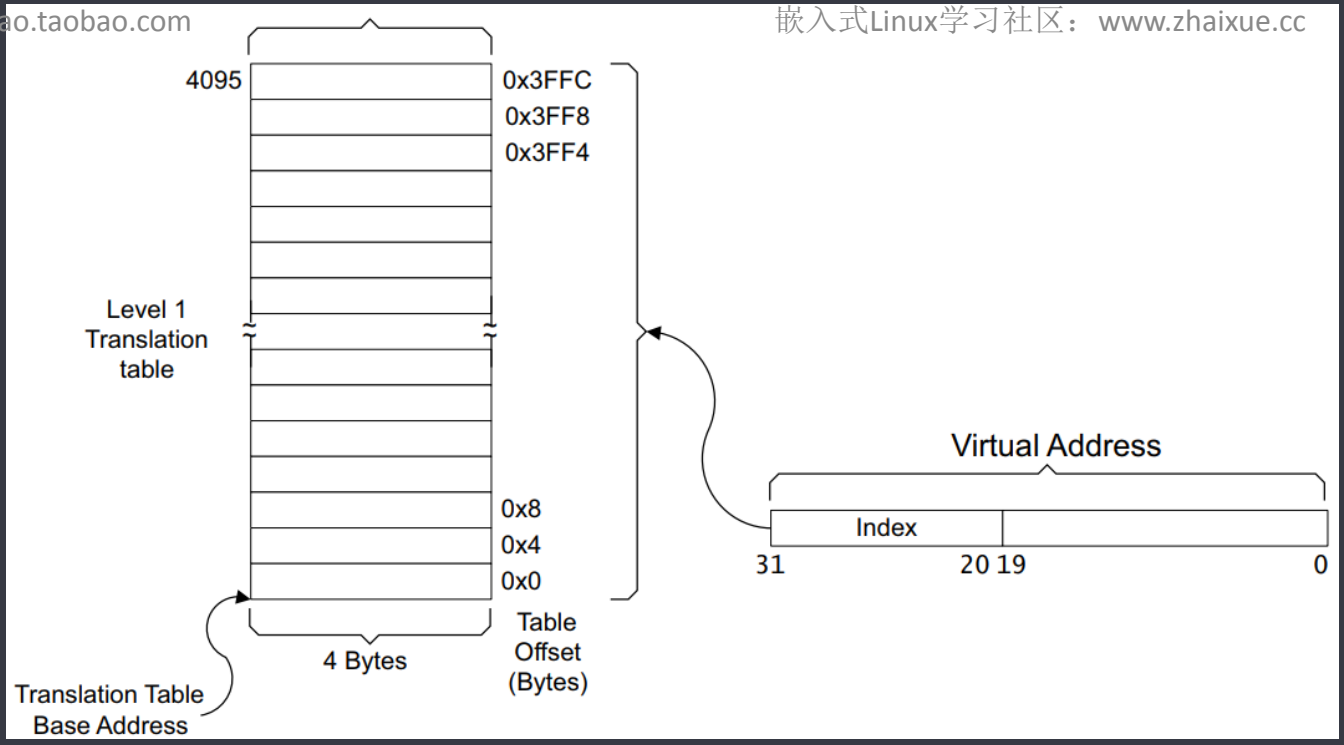
每个二级页表包含256个页表项entry，大小1KB

所有的二级页表大小：一共是4K*1KB=4MB

17 揭开页表神秘的面纱（上）

- 页表深度探析
  - 页表存储在内存的什么地方？
  - 格式到底是怎样的？
  - 不同的映射方式(section/page)，页表有什么变化？
  - 页表的初始化过程分析
  - 新建一个页表项的过程分析

- section



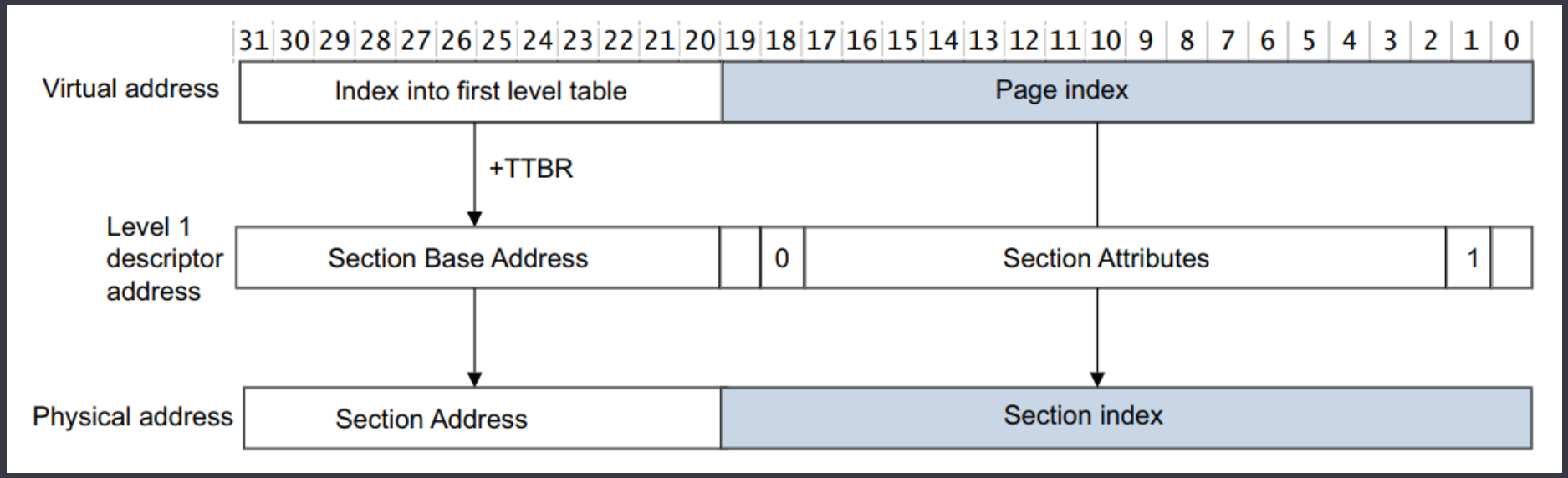
一级页表：

以4KB物理页为映射单位，一个进程4GB的虚拟地址空间需要：
4GB/4KB = 1MB个页表项，每个页表项占用4个字节
每个一级页表需要4MB的存储空间
每个进程需要4MB的内存存储页表，100个进程需要400MB

一级页表：

以1MB为映射单位，每个进程4GB的虚拟地址空间需要：
4GB/1MB=4K个页表项，每个页表项占用4个字节
每个一级页表需要16KB的存储空间
每个进程需要16KB的内存来存储页表

• 1MB section的页表格式



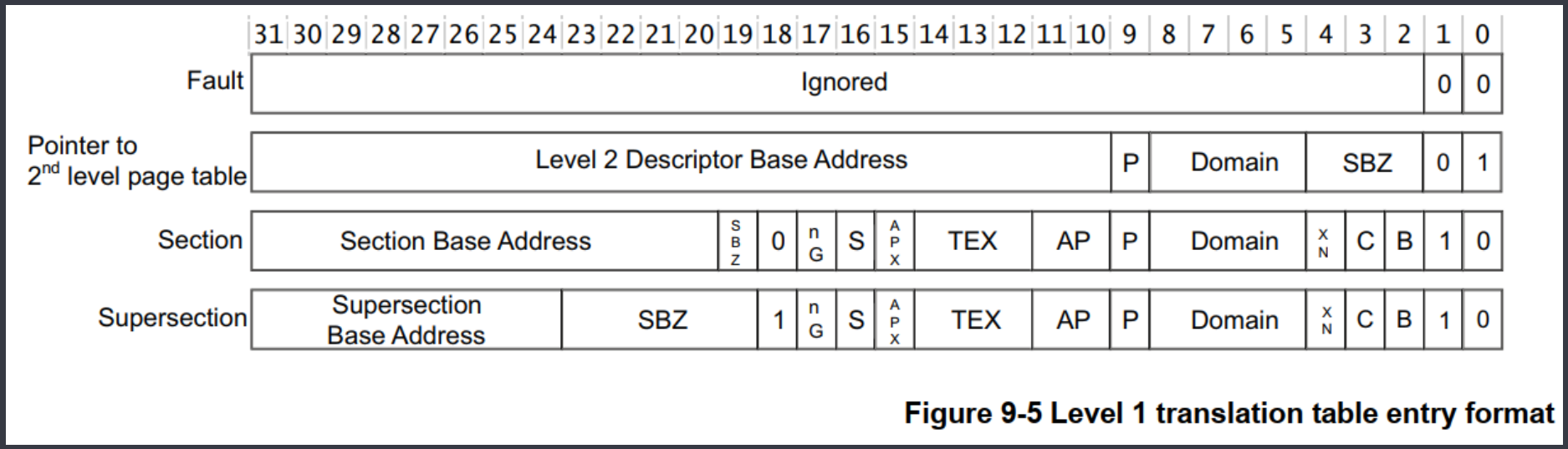


Figure 9-5 Level 1 translation table entry format

• 4KB small page的页表格式

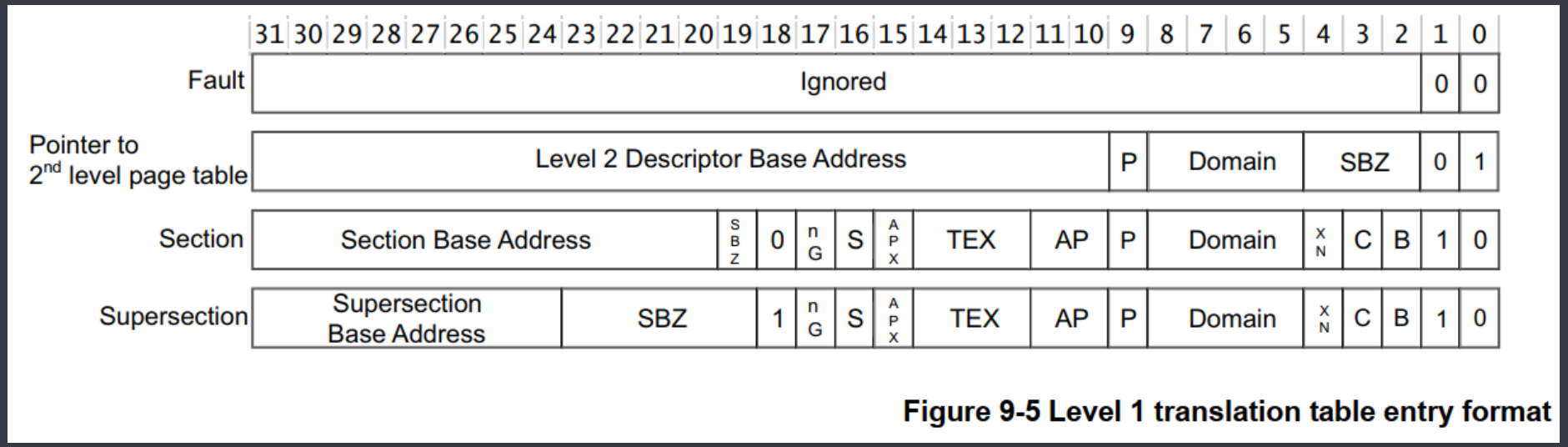


Figure 9-5 Level 1 translation table entry format

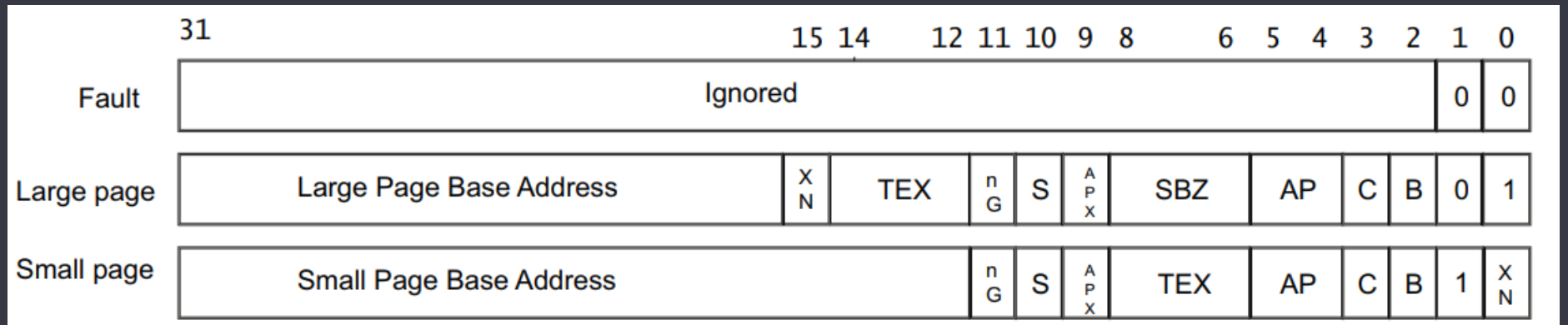


Figure 9-8 Format of a level 2 translation table entry

- 思考
  - 既然section映射可以节省内存
  - 为什么内核启动后还要以page为单位二级映射呢？

18 揭开页表神秘的面纱（下）

页表初探析

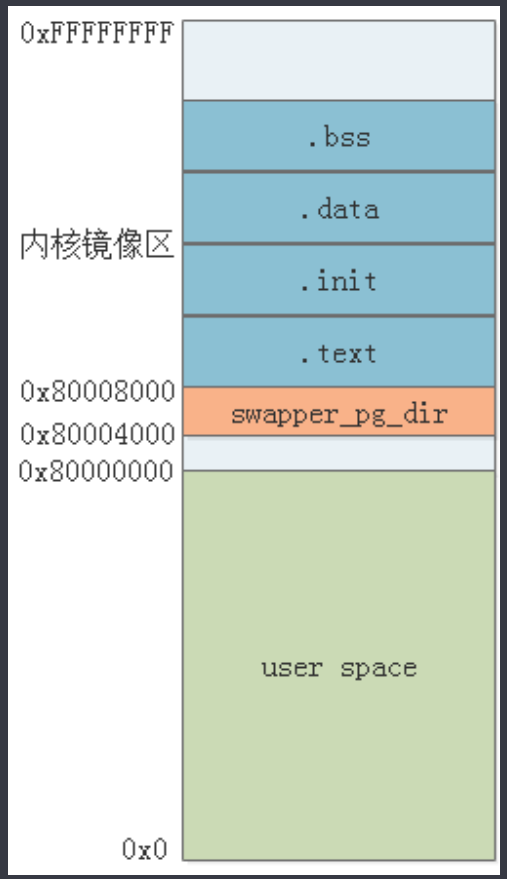
- 使能MMU之前，页表要准备好
- 页表的创建过程分析：__create_page_tables
  - 页表的大小、用途
  - 页表在内存中的地址
  - 页表的创建过程

arch/arm/kernel/head.S:

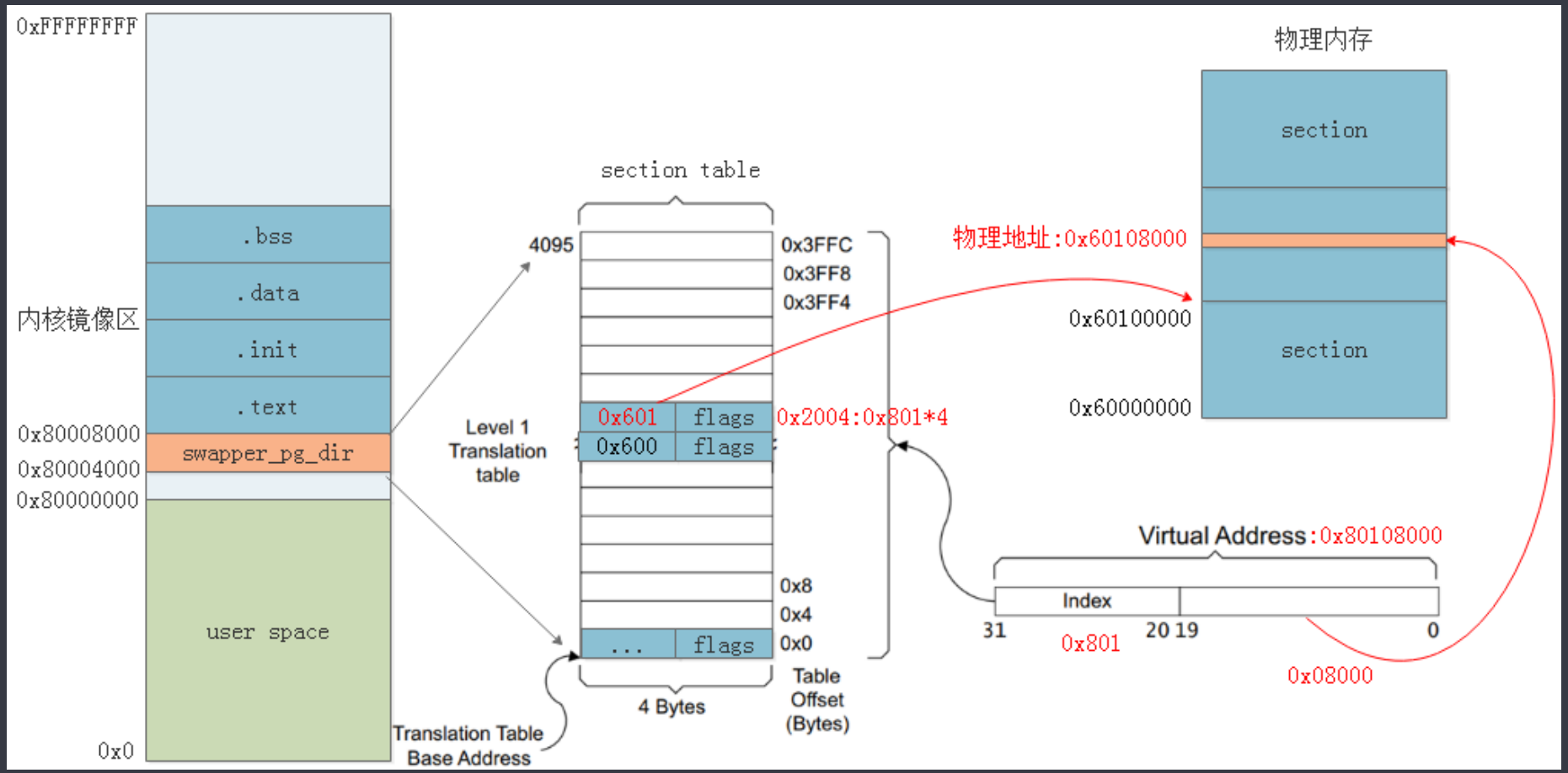
```
.globl      swapper_pg_dir
.equ        swapper_pg_dir, KERNEL_RAM_VADDR - PG_DIR_SIZE

.macro      pgtbl, rd, phys
            add         \rd, \phys, #TEXT_OFFSET
            sub         \rd, \rd, #PG_DIR_SIZE
.endm

1:          orr  r3, r7, r5, lsl #SECTION_SHIFT @ flags + kernel base
            str  r3, [r4, r5, lsl #PMD_ORDER]   @ identity mapping
            cmp  r5, r6  addlo      r5, r5, #1    @ next section
            blo   1b
```



• 一级页表映射：section



- identity mapping

```
        ldr     r7, [r10, #PROCINFO_MM_MMUFLAGS] @ mm_mmuflags
        adr     r0, __turn_mmu_on_loc
        ldmia   r0, {r3, r5, r6}
        sub     r0, r0, r3                      @ virt->phys offset
        add     r5, r5, r0                      @ phys __turn_mmu_on
        add     r6, r6, r0                      @ phys __turn_mmu_on_end
        mov     r5, r5, lsr #SECTION_SHIFT      @__turn_mmu_on对应的section地址
        mov     r6, r6, lsr #SECTION_SHIFT

1:      orr     r3, r7, r5, lsl #SECTION_SHIFT
        str     r3, [r4, r5, lsl #PMD_ORDER]    @ identity mapping
        cmp     r5, r6      addlo     r5, r5, #1    @ next section
        blo     1b
```

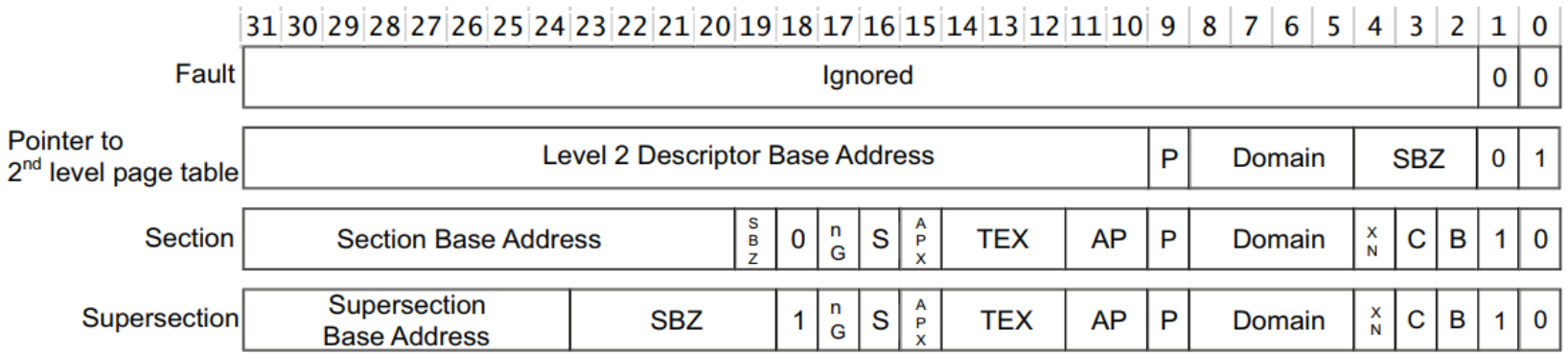


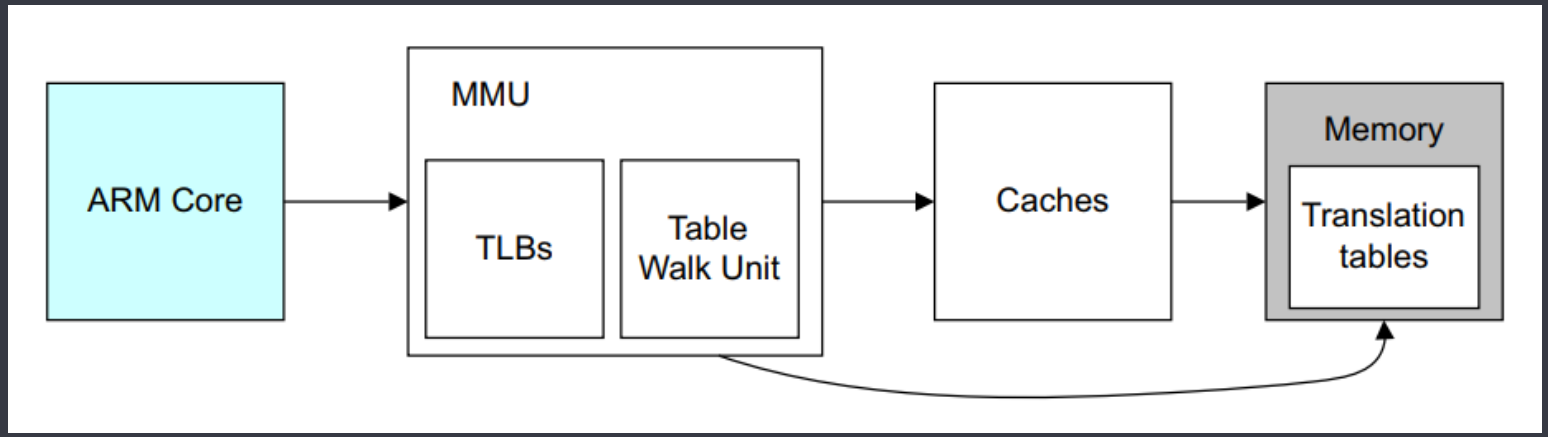
Figure 9-5 Level 1 translation table entry format

• 内核镜像区域的内存映射

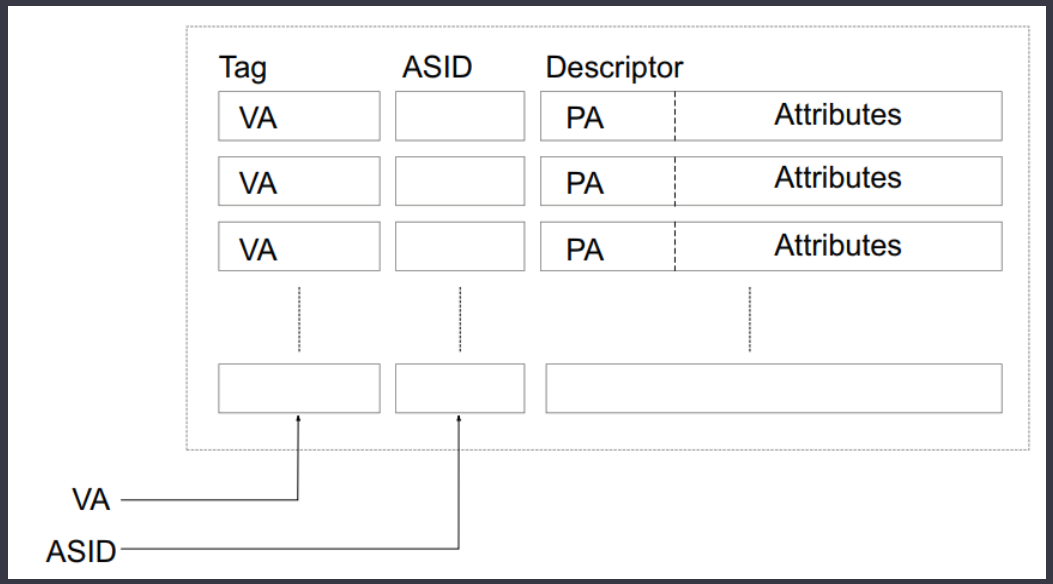
```
/*
* Map our RAM from the start to the end of the kernel .bss section.
*/
        add     r0, r4, #PAGE_OFFSET >> (SECTION_SHIFT - PMD_ORDER)
        ldr     r6, =(_end - 1)
        orr     r3, r8, r7
        add     r6, r4, r6, lsr #(SECTION_SHIFT - PMD_ORDER)
1:      str     r3, [r0], #1 << PMD_ORDER
        add     r3, r3, #1 << SECTION_SHIFT
        cmp     r0, r6
        bls     1b
```

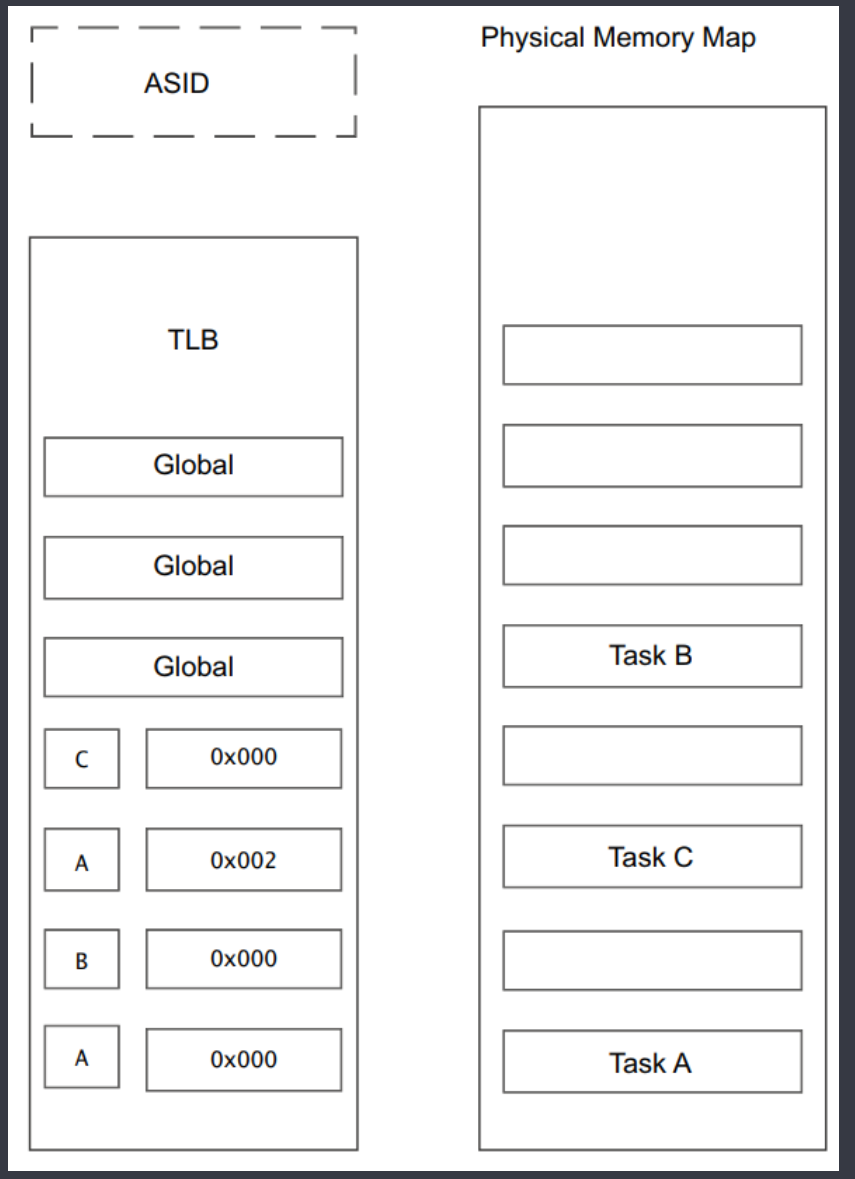
19 TLB 和 Table Walk Unit

- TLB简介
  - The Translation Lookaside Buffer： 页表的缓存
  - Table Walk Unit： 读取内存中的页表到TLB
  - 页表的地址：通过软件写入MMU寄存器中
  - MMU的作用
  - CPU访问虚拟地址的过程：TLB hit、 TLB miss



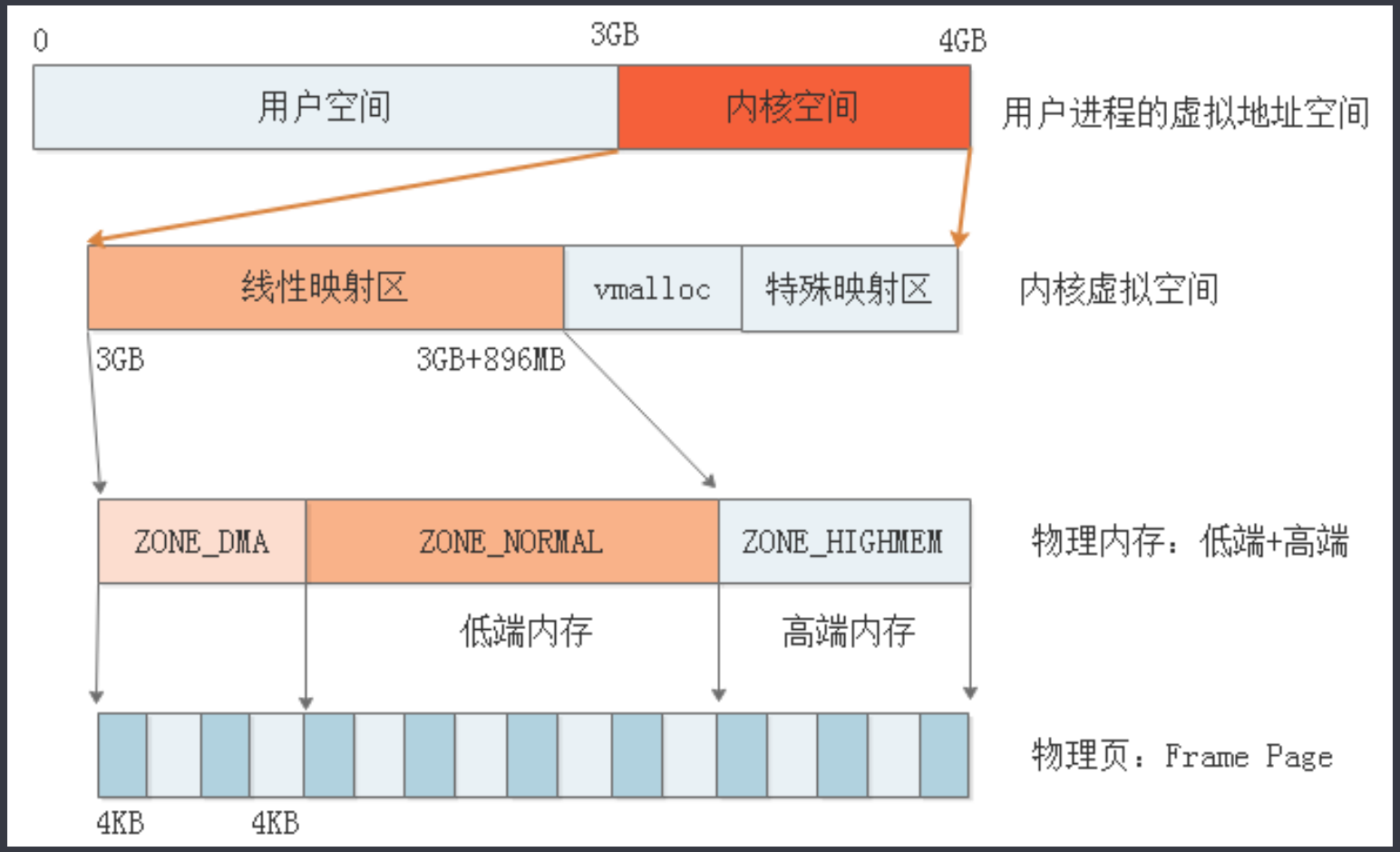
- TLB 工作机制
  - TLB coherency
  - ASID： address space ID
  - CP15 operation for flush：
    - flush_tlb_all
    - flush_tlb_range





20 Linux虚拟内存管理

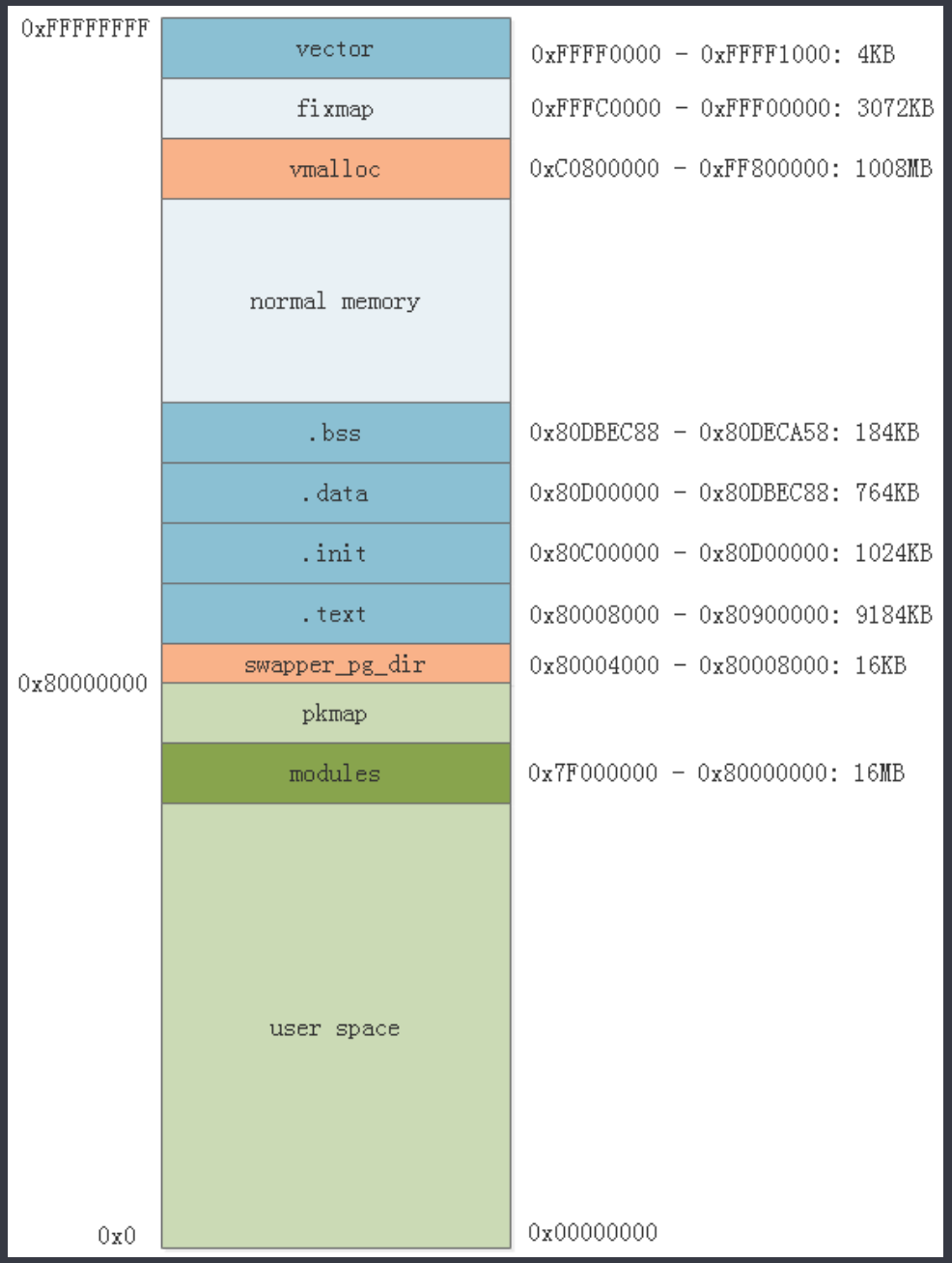
• 32位X86系统下的虚拟内存经典布局



- 虚拟内存划分
  - 用户空间和内核空间划分
    - 3:1
    - 2:2
    - 1:3
    - 思考：为什么会有不同的划分比例？
  - 内核虚拟空间划分：
    - 线性映射区
    - vmalloc
    - fixmap
    - pkmap
    - modules

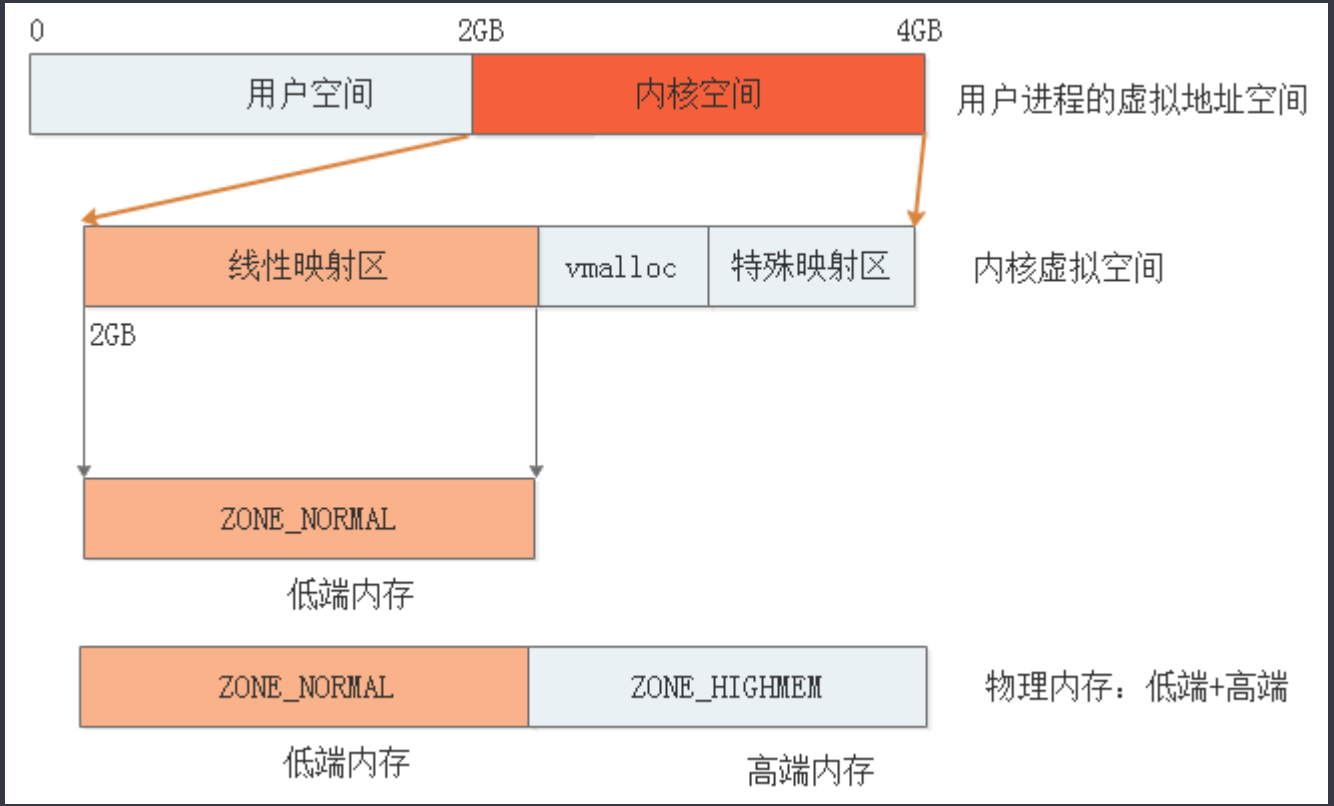
- ARM 32下的内存布局
  - 可配置选项
  - 实验：
    - Vexpress ARM平台
    - 给5.10.x内核打patch
    - 打印各个区域和大小

思考：为什么要将虚拟内存空间划分为不同的区域？



21 虚拟内存管理：线性映射区

- 线性映射区
  - PAGE_OFFSET
  - PHYS_OFFSET



- 线性映射区
  - 地址转换

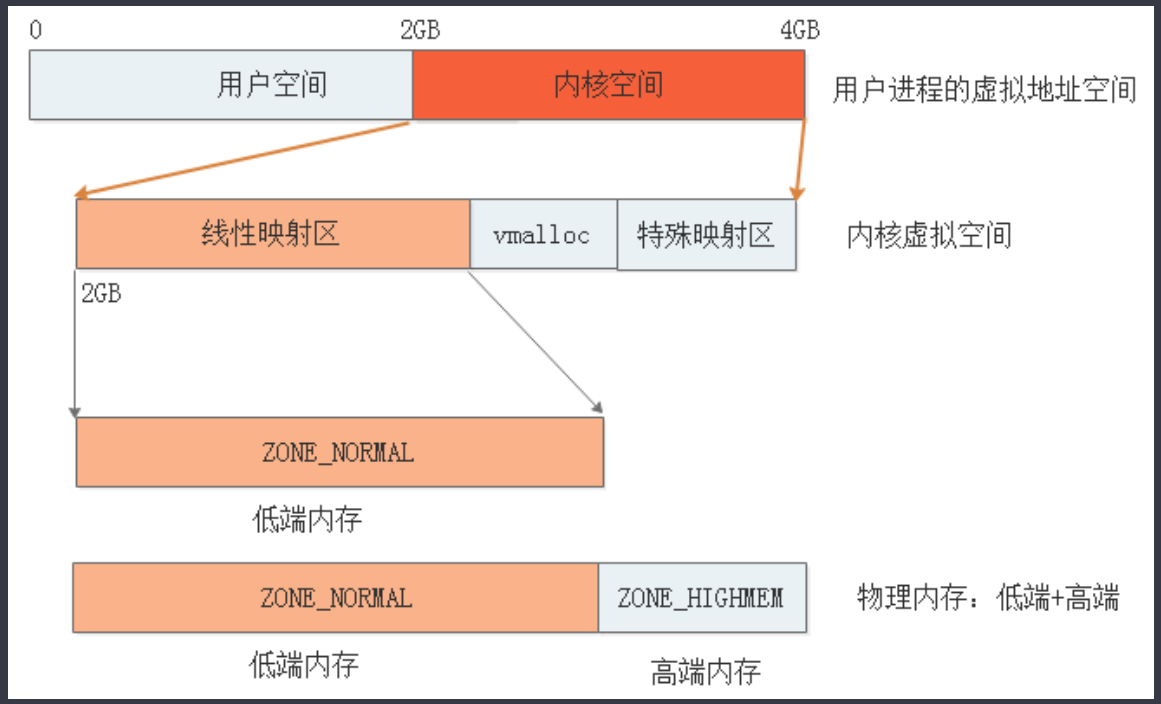
```
[arch/arm/include/asm/memory.h]
#define __pa(x)      __virt_to_phys((unsigned long)(x))
#define __va(x) ((void *)__phys_to_virt(phys_addr_t)(x))

static inline phys_addr_t __virt_to_phys(unsigned long x)
{
        return (phys_addr_t)x - PAGE_OFFSET + PHYS_OFFSET;
}

static inline unsigned long __phys_to_virt(phys_addr_t x)
{
        return x - PHYS_OFFSET + PAGE_OFFSET;
}
```

22 低端内存和高端内存的划分

- 线性映射区的大小
  - 低端内存和高端内存如何划分？
  - PAGE_OFFSET设置不同时，会影响划分吗？
  - 物理内存1GB，对应的虚拟内存布局是怎样的？
  - 物理内存3GB，对应的虚拟内存布局是怎样的？



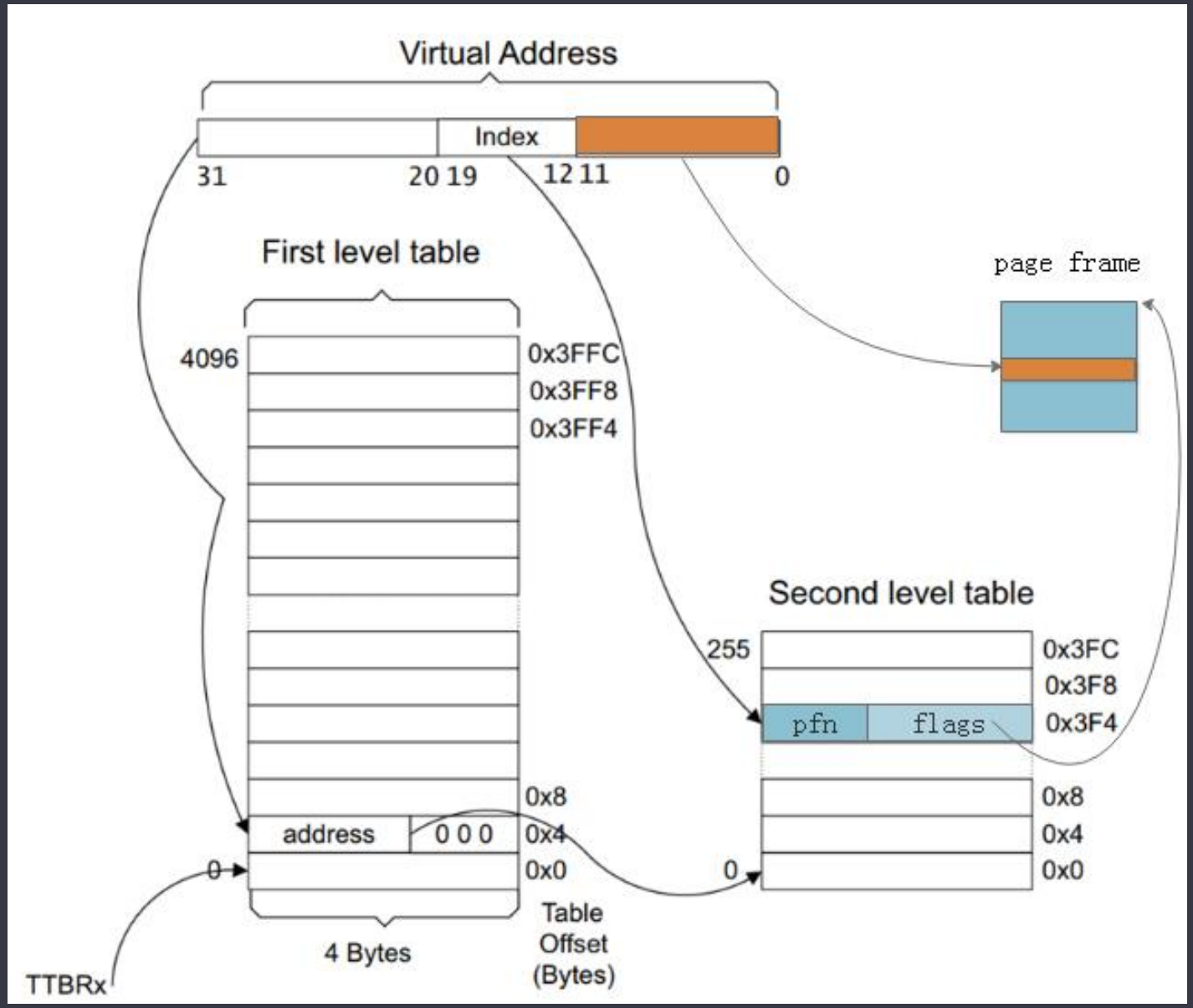
- 实验
  - PAGE_OFFSET为0x80000000，DDR 256MB
  - PAGE_OFFSET为0x80000000，DDR 512MB
  - PAGE_OFFSET为0x80000000，DDR 1GB
  - PAGE_OFFSET为0xC0000000呢？

- 小结
  - 线性映射区大小，和PAGE_OFFSET和物理内存相关
  - 3GB/1GB划分，[3G, 3G+760MB]为线性映射区范围
  - 2GB/2GB划分，[2G, 2G+1760MB]为线性映射区范围
  - 64位系统虚拟空间足够大，全部映射到物理内存
  - ARM 32/64正渐渐舍弃高端内存的概念…

```
#define VMALLOC_OFFSET                  (8*1024*1024)
#define VMALLOC_END                     0xff800000UL
void *  vmalloc_min = (void *)(VMALLOC_END - (240 << 20) - VMALLOC_OFFSET);
vmalloc_limit = (u64)(uintptr_t)vmalloc_min - PAGE_OFFSET + PHYS_OFFSET;
```

23 二级页表的创建过程分析（上）

- 二级页表的初始化


Virtual Address
Index
31
20 19
12 11
0
First level table
page frame
4096
0x3FFC
0x3FF8
0x3FF4
Second level table
255
0x3FC
0x3F8
pfn
flags
0x3F4
0x8
address
0 0 0
0x4
0
0x0
0
0x8
0x4
0x0
4 Bytes
Table
Offset
(Bytes)
TTBRx

• 4KB small page的页表格式

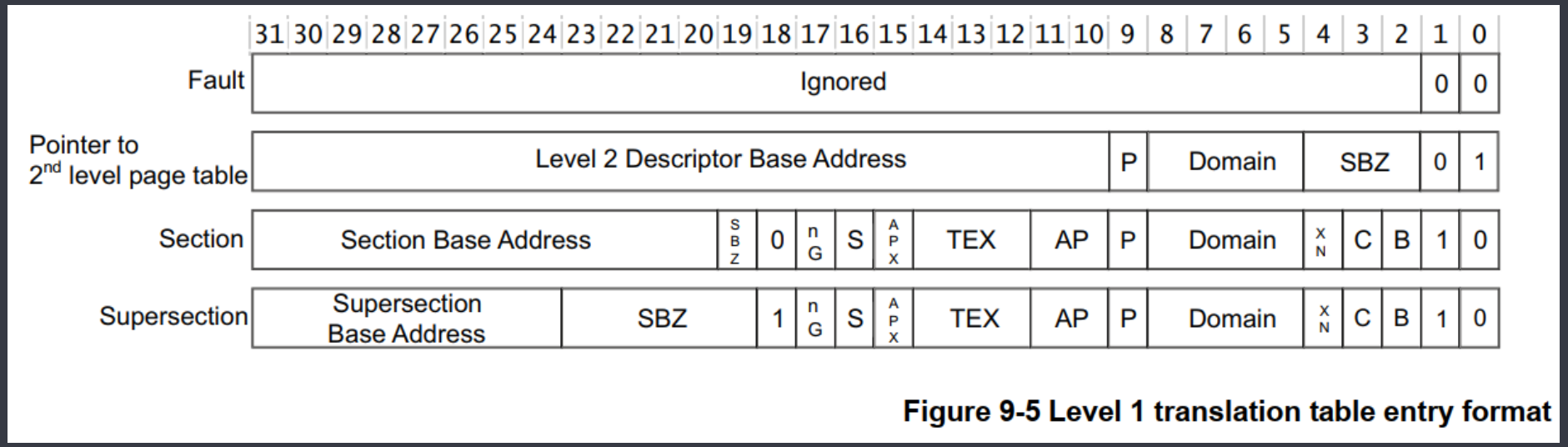


Figure 9-5 Level 1 translation table entry format

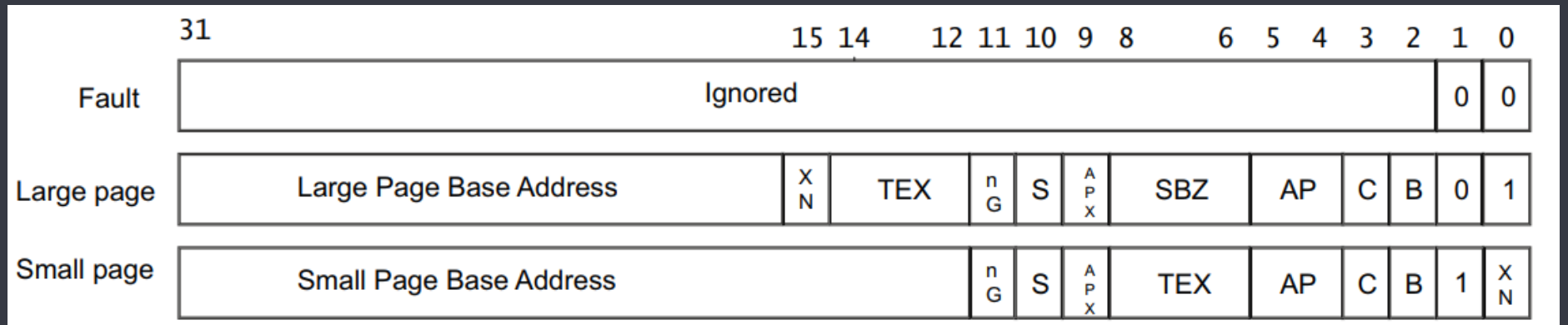
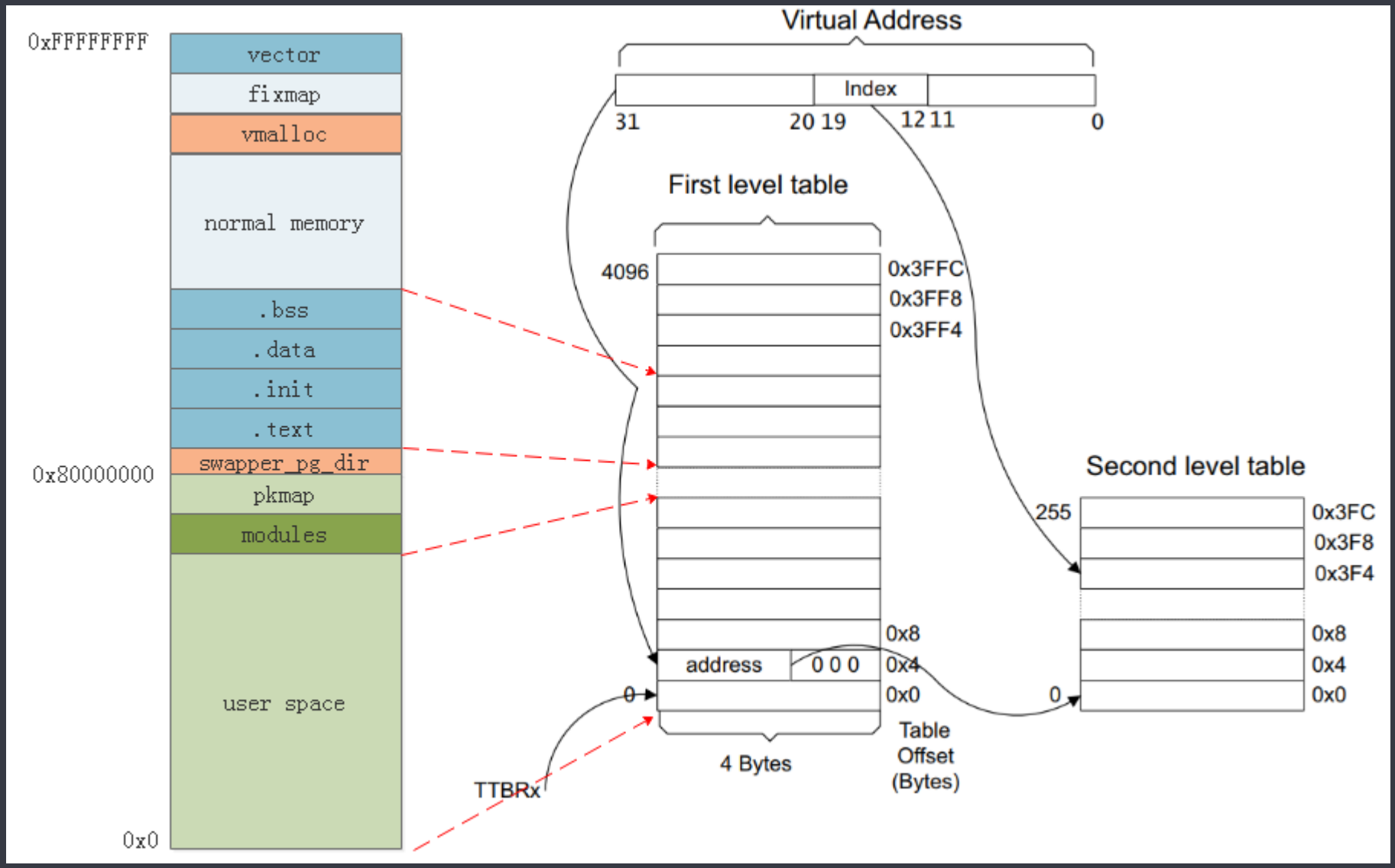


Figure 9-8 Format of a level 2 translation table entry

• 从section映射到page映射



24 二级页表的创建过程分析（中）

- 内存中的页表类型
  - 前提： memblock已初始化
  - 前提： 页表的地址存放在哪里？
  - ARM PTE： ARM MMU硬件页表
  - Linux PTE： 内核使用的软件页表
  - 硬件页表与软件页表如何协同工作？

Linux PTE[n]
Linux PTE[n+1]
ARM PTE[n]
ARM PTE[n+1]

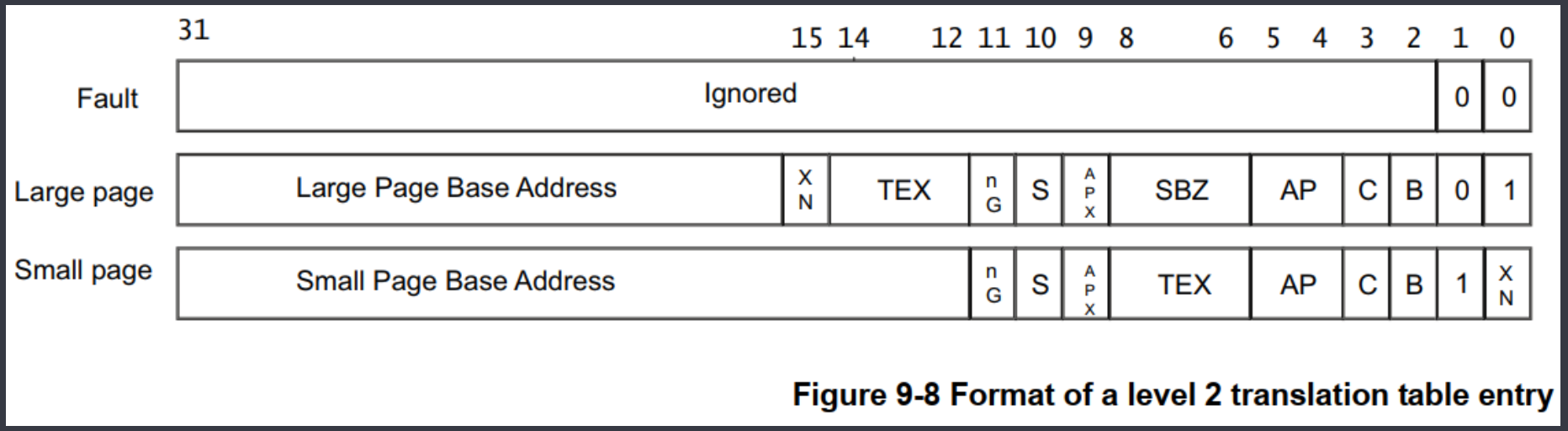
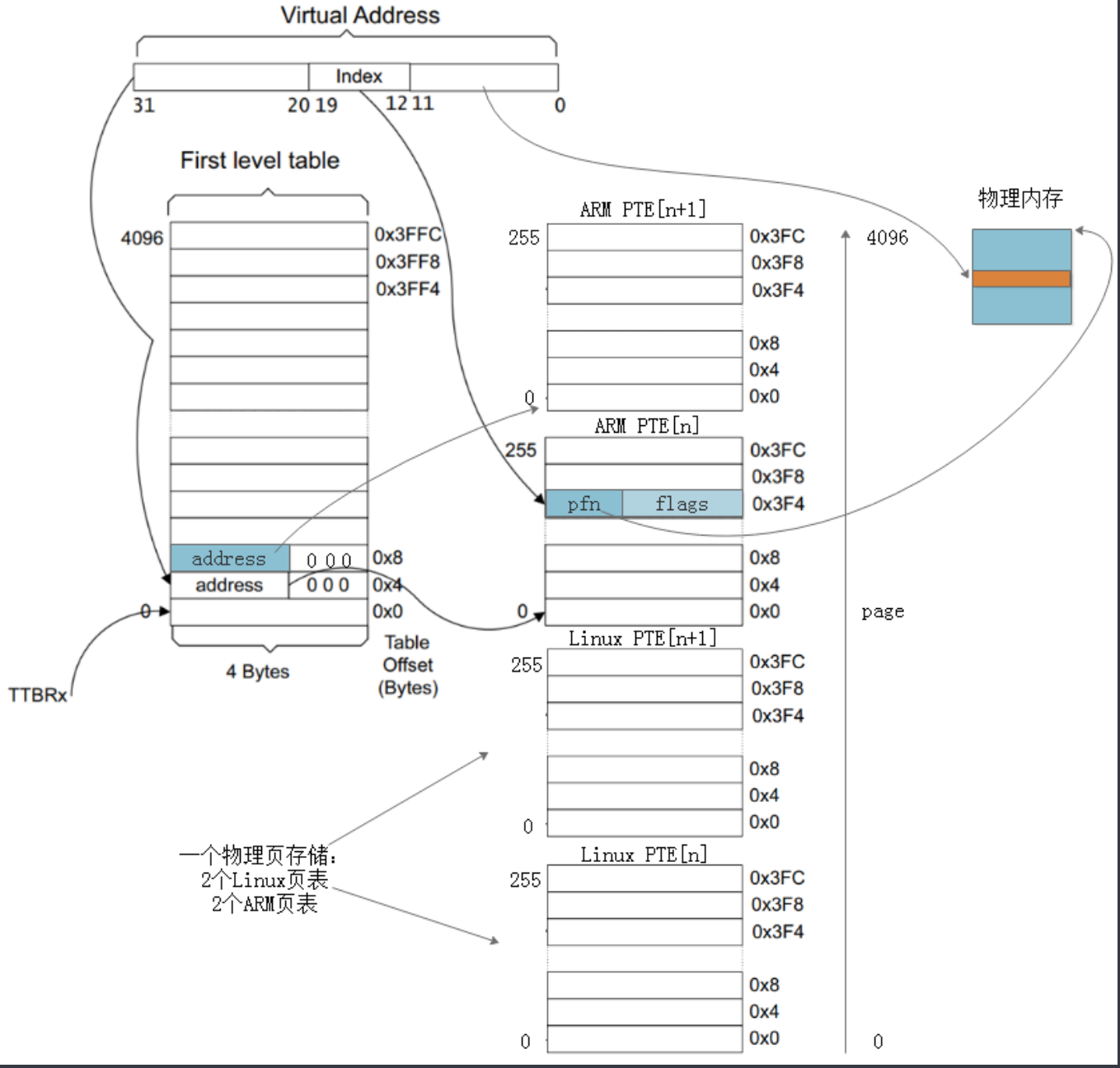


Figure 9-8 Format of a level 2 translation table entry

- 内存中的页表全景图

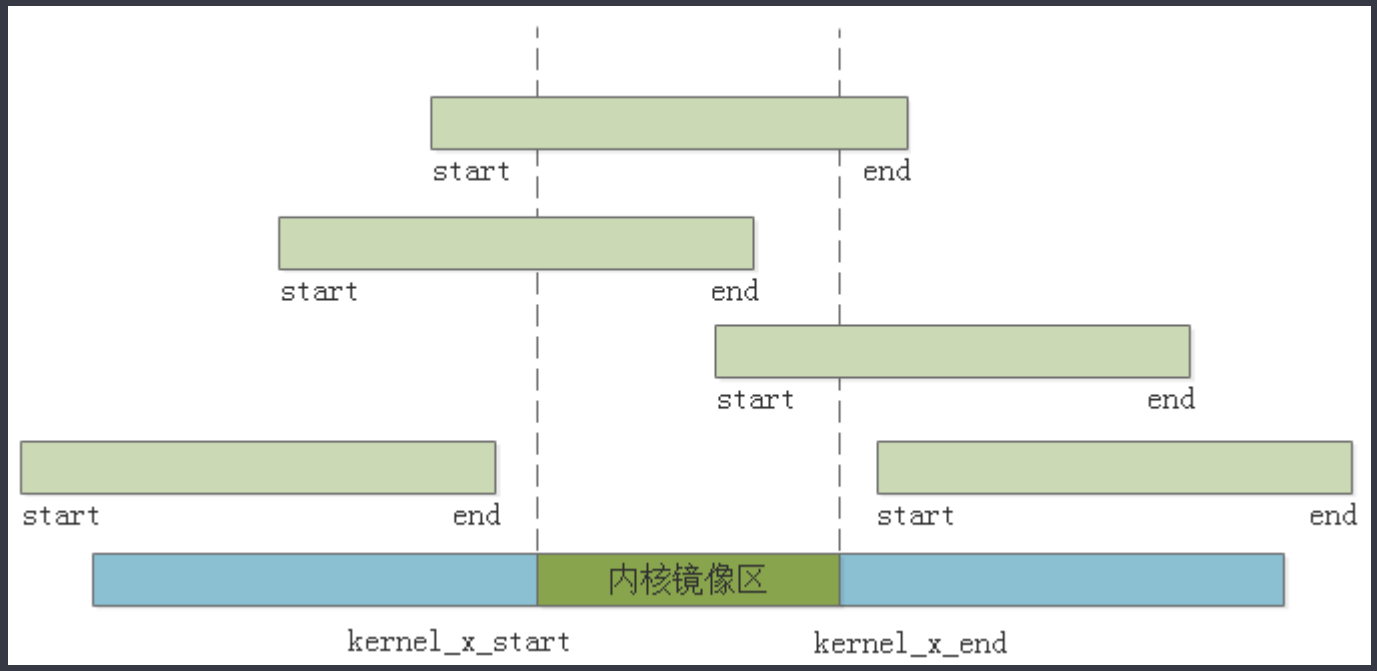
思考：页表的存放为什么要这么安排？



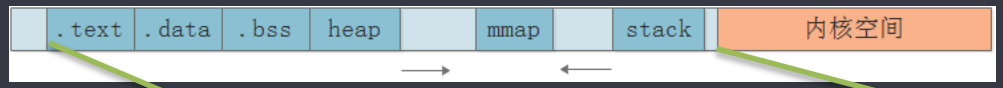
25 二级页表的创建过程分析（下）

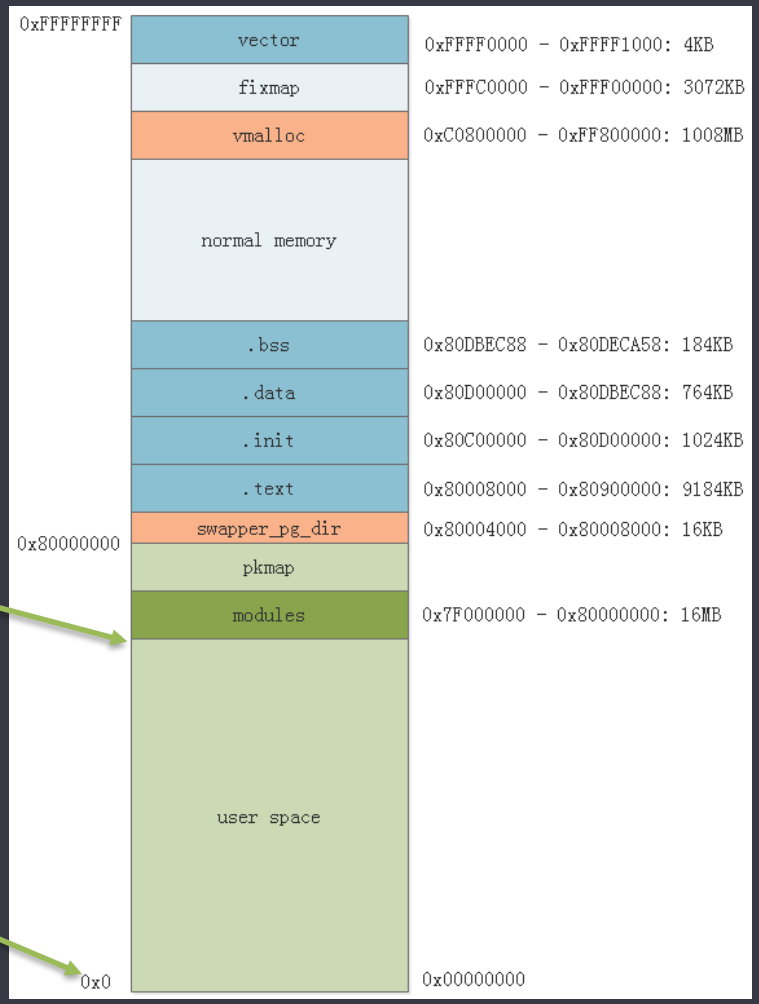
- 页表的创建过程分析
  - 前提：memblock已初始化
  - 前提：一级PGD页表的地址
  - 源码分析：create_mapping
  - 源码分析：cpu_set_pte_ext
  - 难点分析：内核镜像区的重映射

```
start_kernel ->setup_arch
        ->adjust_lowmem_bounds
        ->paging_init
            ->map_lowmem
            ->kmap_init
```



- 对页表的认知刷新
  - 虚拟地址决定了什么？
  - 物理地址决定了什么？
  - 长度决定了什么？





26 虚拟内存管理：vmalloc区

- vmalloc区
  - kmalloc vs vmalloc
  - 内存申请接口：vmalloc/vfree
  - VMALLOC_START到VMALLOC_END之间的一段区域
  - 虚拟空间是连续的，物理空间可以不连续
  - vmalloc主要用在什么地方？
  - vmalloc最大能申请多大的内存
  - vmalloc区的大小怎么计算？默认大小呢？
  - vmalloc区域的页表映射

- vmalloc编程接口
  - 编程示例：使用vmalloc申请和释放内存
  - vmalloc实现机制分析
    - 从VMALLOC_START到VMALLOC_END查找一片虚拟地址空间
    - 根据内存的大小从伙伴系统申请多个物理页帧page
    - 把每个申请到的物理页帧逐页映射到虚拟地址空间

```
void *vmalloc(unsigned long size);
void *vzalloc(unsigned long size);
void vfree(const void *addr);
unsigned long vmalloc_to_pfn( const void *vmalloc_addr);
struct page *vmalloc_to_page( const void *vmalloc_addr);
```

- vmalloc 分配器
  - 核心数据结构：vmap_area、vm_struct
  - 全局变量：vmap_area_root、vmap_area_list

```
struct vm_struct {
        struct vm_struct        *next;
        void                    *addr;      // 当前申请的vmalloc区域的起始地址
        unsigned long           size;       // 当前申请的vmalloc区域的大小
        unsigned long           flags;
        struct page             **pages;    //使用数组保存所有映射的物理页page，pages指向该数组
        unsigned int    nr_pages;           // vmalloc映射到物理内存对应的page的个数
        phys_addr_t     phys_addr;          // 设备的IO内存映射内存地址，普通内存为0
        const void      *caller;            // 调用vmalloc函数的caller
};
struct vmap_area {  //表示vmalloc申请的一片内存区域，使用vmalloc返回的虚拟地址空间
        unsigned long va_start;     // vmalloc申请的虚拟地址空间起始地址
        unsigned long va_end;       // vmalloc申请的虚拟地址空间结束地址
        struct rb_node rb_node;     /* address sorted rbtree */ // 添加到vmap_area_root红黑树上
        struct list_head list;      /* address sorted list */   // 添加到vmap_area_list链表上
        union {
                unsigned long subtree_max_size; /* in "free" tree */
                struct vm_struct *vm;           /* in "busy" tree */
                struct llist_node purge_list;   /* in purge list */
        };
}
```

27 寄存器映射：ioremap

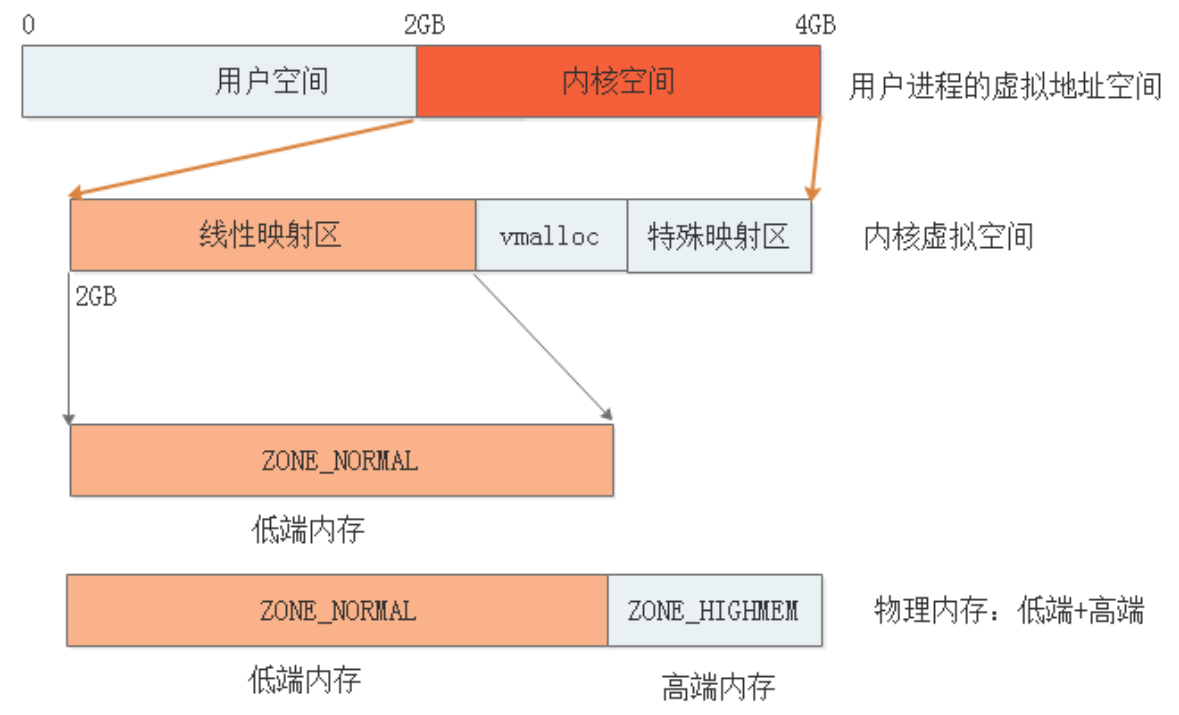
- 寄存器的ioremap
  - I/O端口与I/O内存
  - I/O-mapped与Memory-mapped
  - Linux驱动中寄存器为什么要ioremap？
  - ioremap实现机制分析

```
//arch/arm/mm/ioremap.c
void __iomem *ioremap(resource_size_t res_cookie, size_t size);
   ioremap -> arch_ioremap_caller -> __arm_ioremap_caller
void iounmap(volatile void __iomem *cookie);

//io.h
static inline void __iomem *ioremap(phys_addr_t addr, size_t size);
                -> ioremap_prot
void iounmap(volatile void __iomem *addr);
```

28 高端内存映射

- 高端物理内存
  - 为什么会有高端内存区域？
  - 配置：CONFIG_HIGHMEM
  - 配置： memory split
  - 什么情况下，才会有ZONE_HIGHMEM？
  - 高端内存的初始化



- 再一次讨论vmalloc
  - 当只有低端物理内存时，vmalloc从哪里申请内存？
  - vmalloc建立的映射是否和线性映射冲突？
  - 当有高端内存时，vmalloc从哪里申请物理内存？
  - vmalloc源码再分析
  - vmalloc扮演的角色

29 虚拟内存管理：pkmap区

- pkmap区
  - 编程接口函数：
    - void *kmap (struct page *page);
    - void kunmap (struct page *page);
    - void *kmap_atomic (struct page *page);
  - 编程实验：
    - 在低端内存申请一个物理页帧，使用kmap建立映射
    - 在高端内存申请一个物理页帧，使用kmap建立映射
    - 内核配置选项：CONFIG_HIGHMEM
    - 打印出映射后的虚拟地址，观察虚拟地址变化
  - 使用注意事项

- pkmap区
  - pkmap实现机制
  - 分配表： int pkmap_count[512];
  - page_address_htable[128]
  - pkmap_page_table[512]

```
#define LAST_PKMAP                      PTRS_PER_PTE
#define PTRS_PER_PTE                    512
#define PA_HASH_ORDER                   7

struct page_address_map {
                struct page *page;
                void *virtual;
                struct list_head list;
};
struct  page_address_map  page_address_maps[LAST_PKMAP];

static struct page_address_slot {
                struct list_head  lh;                   /* List of page_address_maps */
                spinlock_t lock;                        /* Protect this bucket's list */
} ____cacheline_aligned_in_smp   page_address_htable[1<<PA_HASH_ORDER];
```

30 虚拟内存管理：fixmap区

- fixmap区
  - fixmap的作用
  - fixmap初始化
    - start_kernel->setup_arch ->early_fixmap_init
    - start_kernel->setup_arch -> early_ioremap_setup()

31 虚拟内存管理：modules区

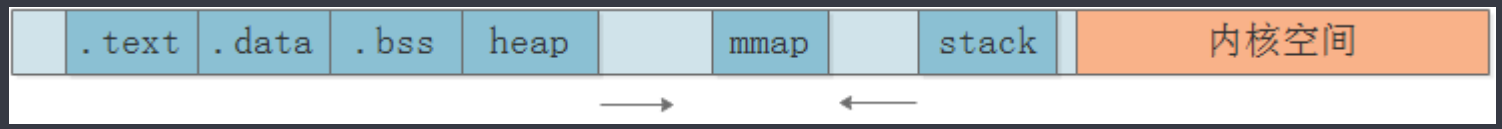
- modules区
  - insmod过程分析
    - busybox：insmod命令调用 sys_init_module
    - 内核中系统调用实现：kernel/module.c: init_module
  - 思考：
    - 为什么insmod最终要到vmalloc区去申请内存？
    - module区的未来何去何从？

```
/*
 * Moudule Virtual Space
 * 0    3G                                          4G
 * +----+----+---------------+-------------------------+-----+
 * |    |    |               |                         |     |
 * |    |    |               |  Module Virtual Space   |     |
 * |    |    |               |                         |     |
 * +----+----+---------------+-------------------------+-----+
 *                           A                         A
 *                           |                         |
 *                           |                         |
 *                           |                         |
 *                           o                         o
 *                      MODULES_VADDR             MODULES_END
 */
```

32 用户进程的页表

• 用户进程的页表

- 用户进程的空间描述：mm_struct、vm_area_struct
- 用户进程的页表与内核的页表



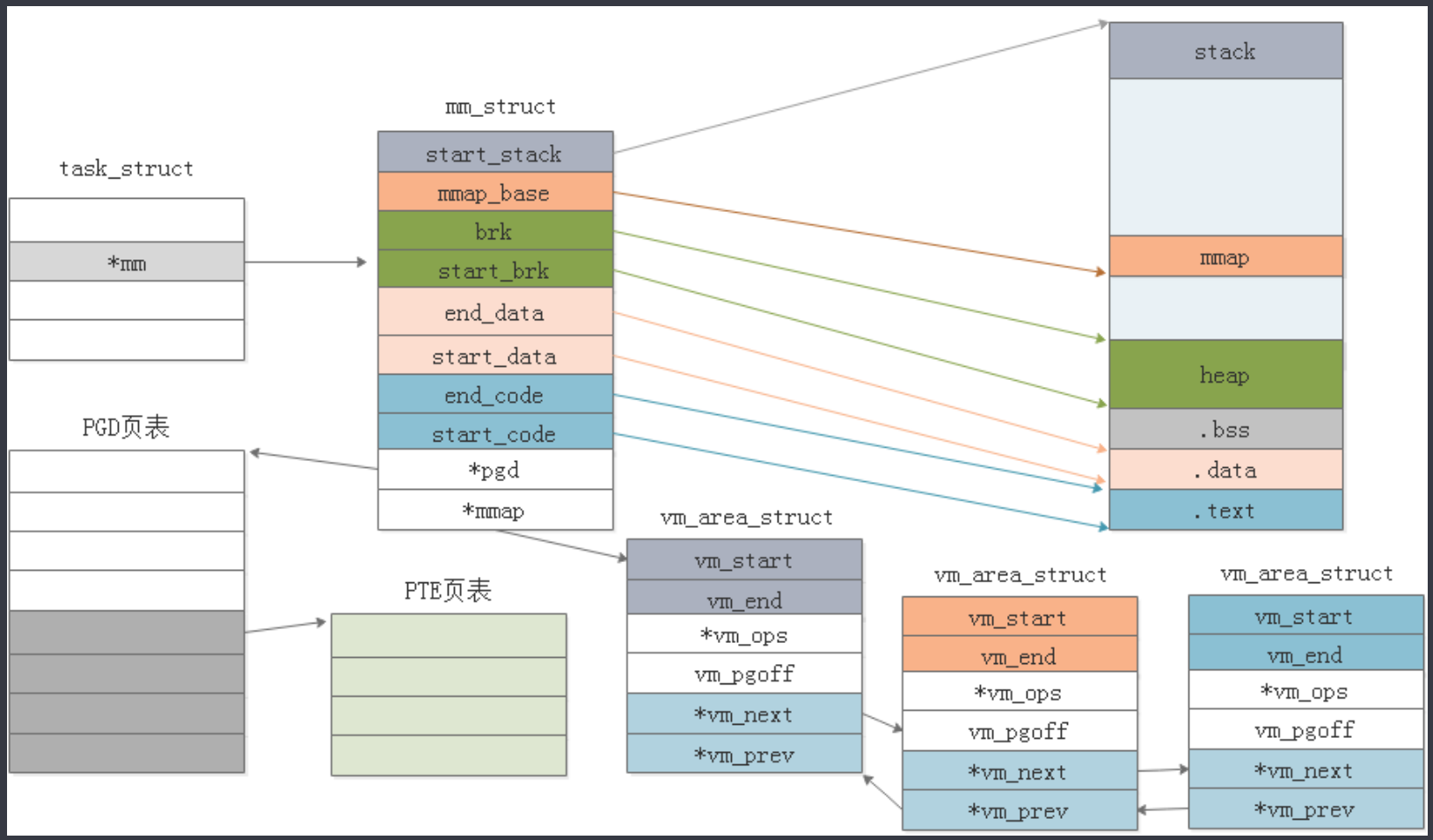

```
struct task_struct {
        struct mm_struct        *mm; //描述一个进程的整个虚拟地址空间
        ...
};

struct mm_struct {
        struct vm_area_struct  *mmap;  //list of VMAs，每个VMA表示一个区域
        pgd_t                   * pgd; // 当前进程的页表基址
};

struct vm_area_struct {
        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, *vm_prev;
};
```

• 用户进程页表的创建过程分析

- 内核源码分析：fork系统调用
- 页表的拷贝
- 设置页保护位



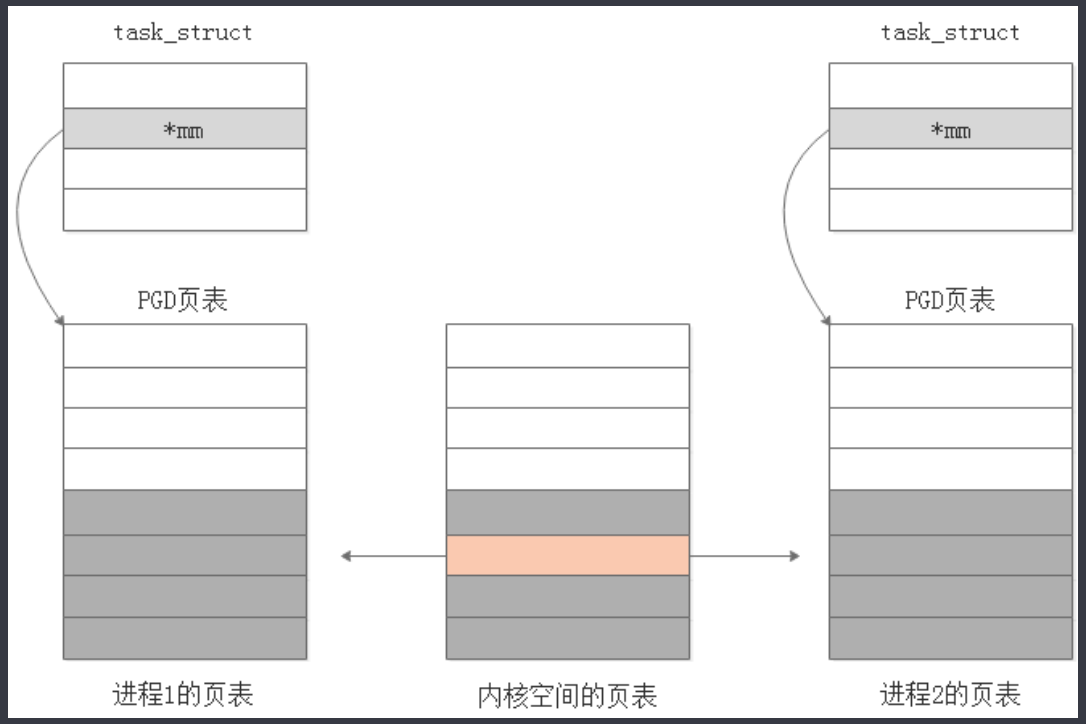
33 缺页异常机制

- 缺页异常
  - 什么是缺页异常？
  - 什么时候会触发缺页异常？
  - 缺页中断回调函数注册
  - 缺页中断处理流程

- 用户进程缺页异常分析
  - 缺页中断流程内核源码分析
  - 新的物理页帧的创建
  - 更新对应的页表entry
  - 写保护权限解除，COW，copy-on-write
  - 父子进程各自运行

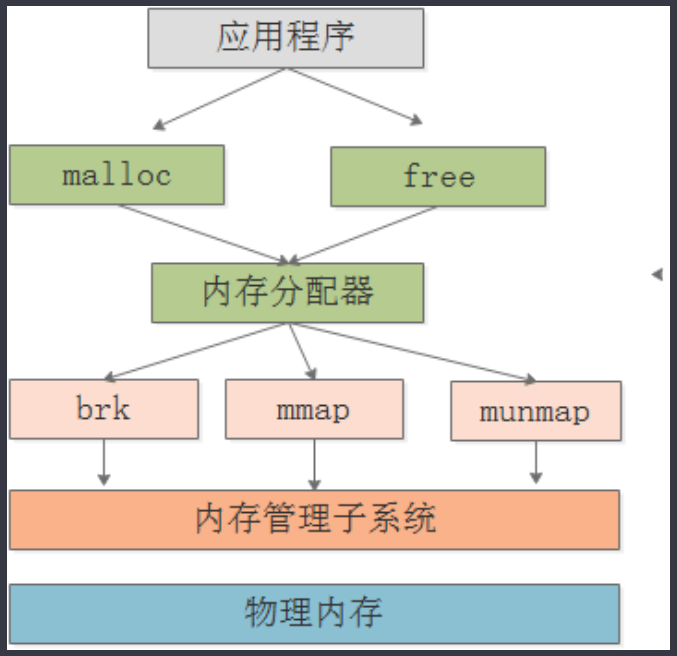
34 用户页表的刷新

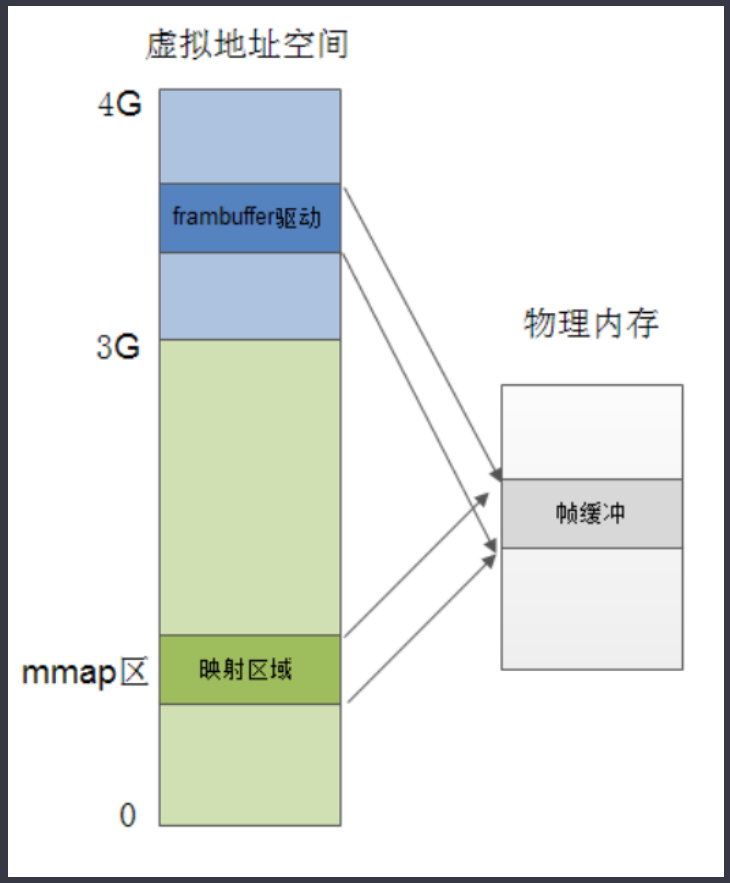
- 页表的同步
  - 用户进程的内核页表
  - 内核空间的内核页表
  - vmalloc/vfree 操作产生的页表更新
  - TLB刷新
  - 缺页异常分析：页表同步更新

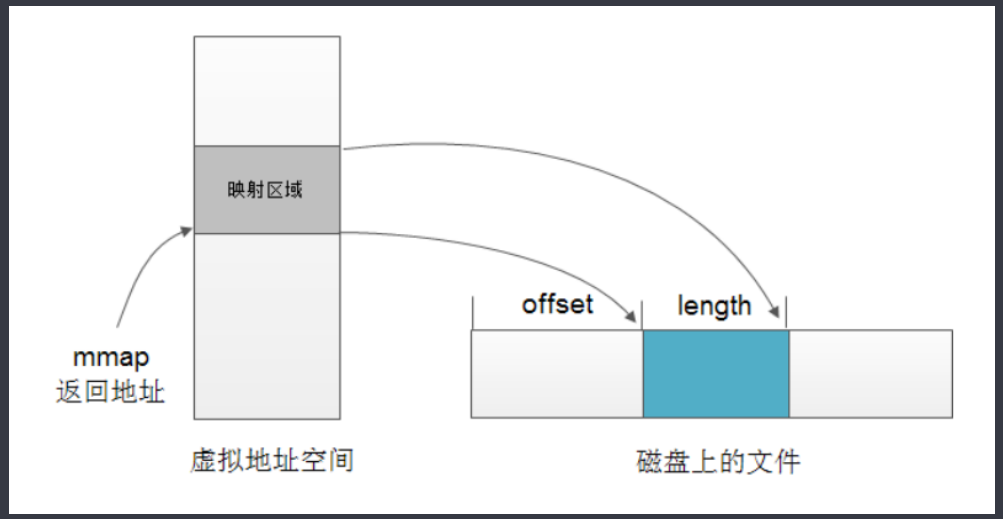


35 mmap映射机制：编程实例

- mmap映射机制
  - 什么是mmap映射？
  - 哪些场合需要mmap内存映射？







- mmap编程示例
  - 将文件映射到虚拟地址空间，通过地址读写文件
  - 实现一个字符设备驱动
    - 实现字符设备的read、write接口
    - 实现字符设备的mmap接口
    - 编写应用程序，通过文件IO接口读写字符设备
    - 编写应用程序，通过mmap接口读写字符设备

36 mmap映射机制：remap_pfn_range

- 用户空间的虚拟地址映射分析
  - 接口函数实现分析：remap_pfn_remap
  - 虚拟地址是如何分配的？
  - mmap系统调用流程分析

```
void *mmap(void *addr, size_t length, int prot, int flags, int fd, off_t offset);
int remap_pfn_range (struct vm_area_struct *vma, unsigned long addr,
                     unsigned long pfn, unsigned long size, pgprot_t prot)
```

- mmap映射类型
  - 私有匿名映射：大块内存分配(malloc > 128KB)
  - 私有文件映射：动态库加载
  - 共享匿名映射：父子进程间通信，共享内存
  - 共享文件映射：设备内存映射，进程间通信

思考：

mmap映射的本质机制是什么？
如何分配对应的虚拟地址？
如何找到对应的物理地址？
映射三要素

37 mmap映射机制：文件映射

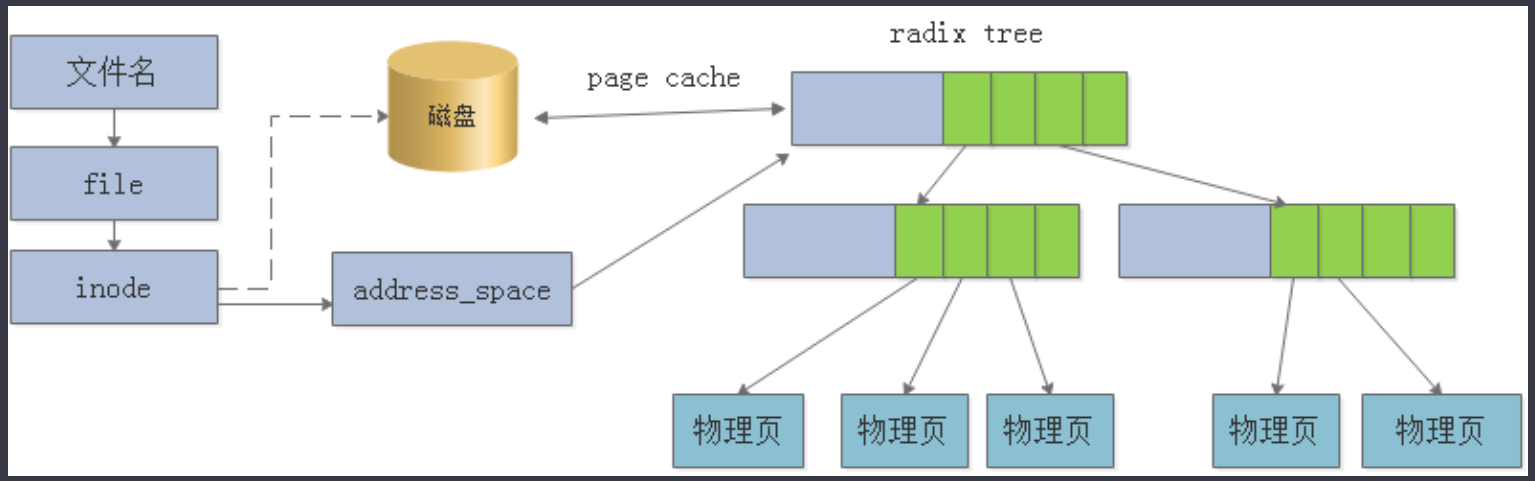
- mmap文件映射
  - 文件映射的底层实现分析

映射三要素：

mmap文件映射的底层机制本质是什么？

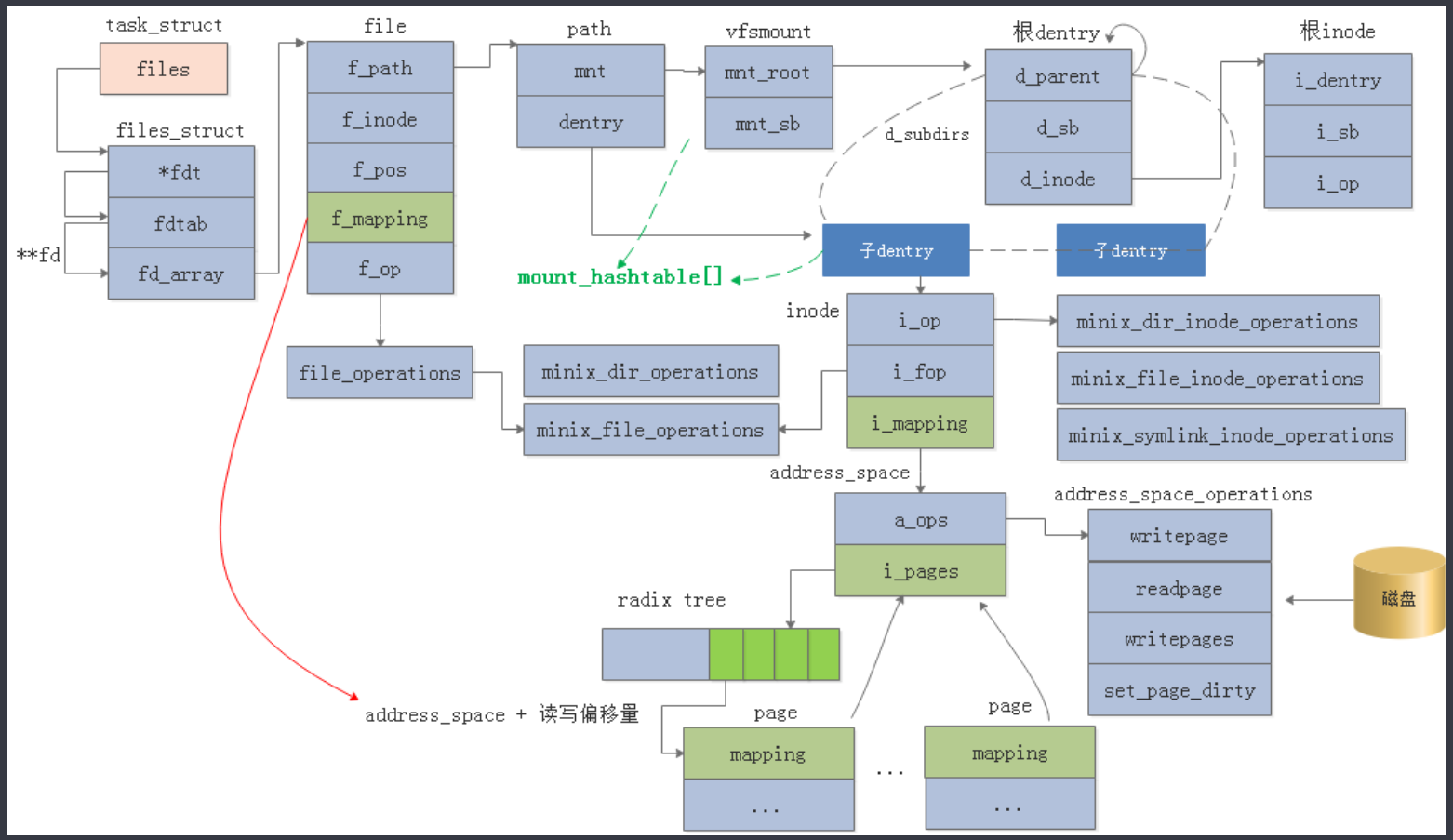
如何分配对应的虚拟地址？

如何找到对应的物理地址？

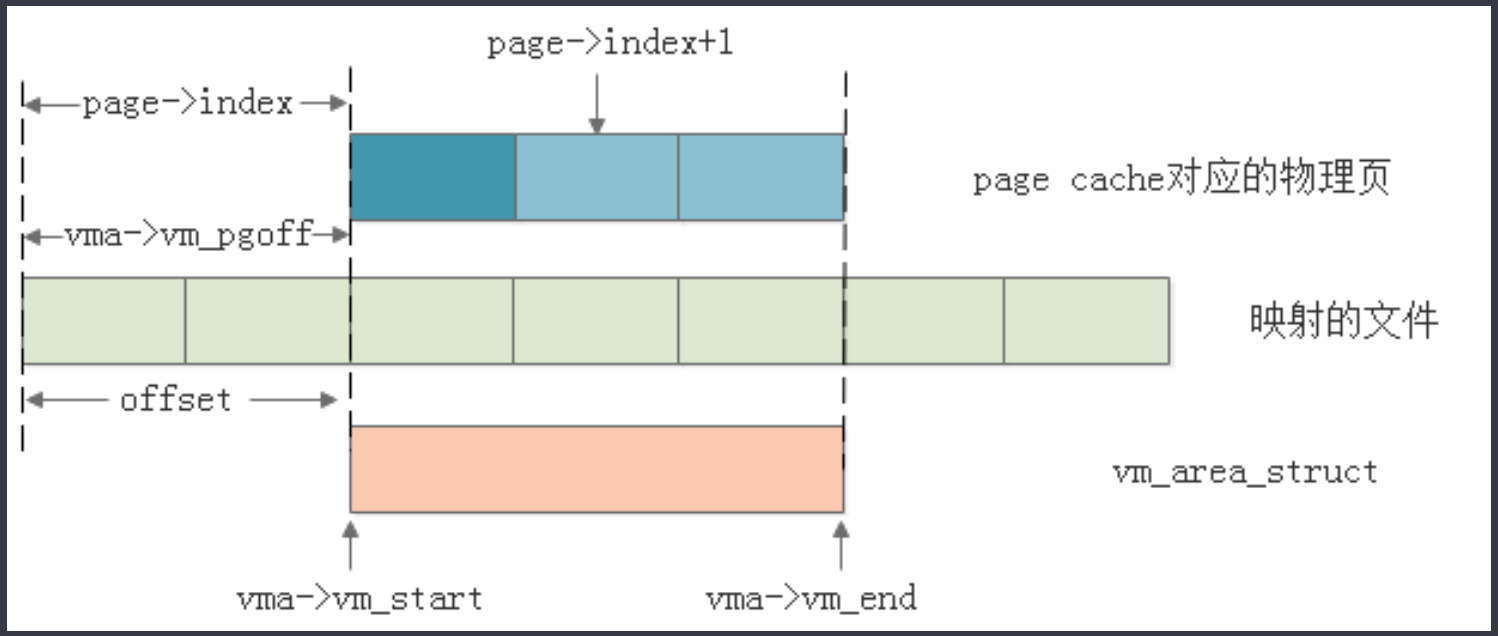


- 用户空间的mmap映射特点
  - 不会在映射时分配物理页
  - 仅仅是建立一种关联：
    - 虚拟地址和文件地址偏移建立关联
    - 设置好缺页异常回调函数
    - 返回分配的虚拟地址给用户程序
  - 读写的时候触发缺页异常
    - 调用回调函数
    - 在异常处理中分配物理页
    - 建立映射，更新页表

- 如何建立关联？
  - address_space与page cache的关联
  - 如何定位page cache？



- 如何建立关联？
  - 虚拟地址和文件偏移地址之间的关联
  - 难点：offset与vma->vm_pgoff的关系



38 文件映射缺页异常

- 文件缺页异常
  - 映射后的逻辑关联：
    - 虚拟地址：vma->vm_start
    - address_space
    - 文件读写偏移：vma->pgoff
    - 异常处理回调：vma->vm_ops
  - 读文件缺页异常分析
    - 读写异常的虚拟地址：vmf
    - 如何查找page：address_space + vma->vm_pgoff
    - 如何建立映射
    - 如何更新页表

- 文件映射过程分析
  - 用户态系统调用：mmap
  - 内核态mmap：file_operations->mmap
  - 核心结构体：vm_area_struct、address_space
  - 缺页异常：handle_pte_fault(&vmf);

39 设备映射缺页异常

- 编程示例
  - 编写一个字符设备驱动
  - 编写测试应用程序
  - 要求：
    - 设备设备文件内存的映射功能
    - 完成设备的mmap功能
    - 完成缺页异常回调函数
    - 编写应用程序，测试mmap功能

vmf在异常处理中扮演色角色：
保存发生异常的虚拟地址vma：vmf->vma=vma
调用在mmap时注册的fault回调函数分配page
保存分配的page地址：vmf->page = page
调用finish_fault建立映射，更新页表

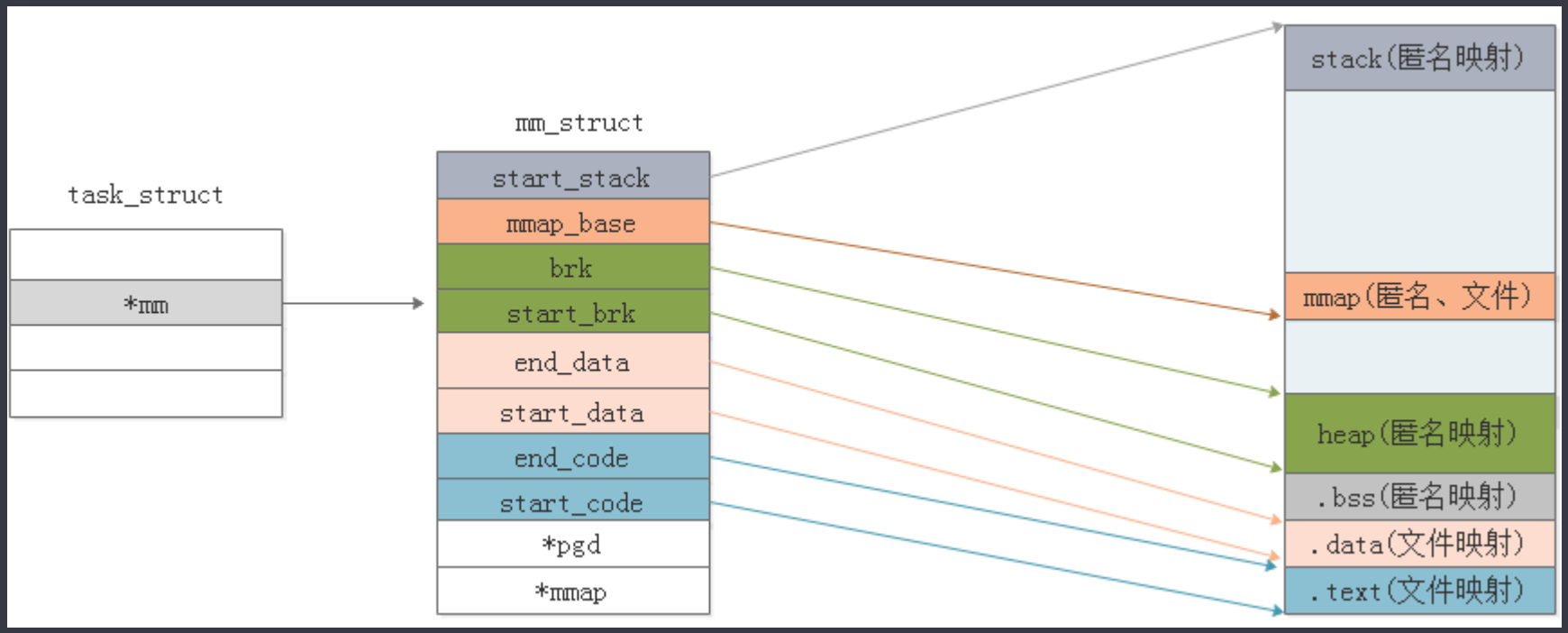
40 mmap映射机制：匿名映射

- 匿名映射
  - 什么是匿名内存？匿名页面，文件页面？
  - 什么是匿名映射？
  - 匿名映射用在什么场合？
    - 使用malloc申请的堆内存
    - 使用mmap创建的内存匿名映射(注意参数fd变化)
    - 如何判断匿名映射？vma->vm_ops
  - 核心结构体
    - struct anon_vma
    - struct anon_vma_chain
    - struct vm_area_struct

- malloc实现机制分析
  - 匿名映射过程分析
  - 匿名页面缺页异常处理过程
  - empty_zero_page
  - do_anonymous_page流程

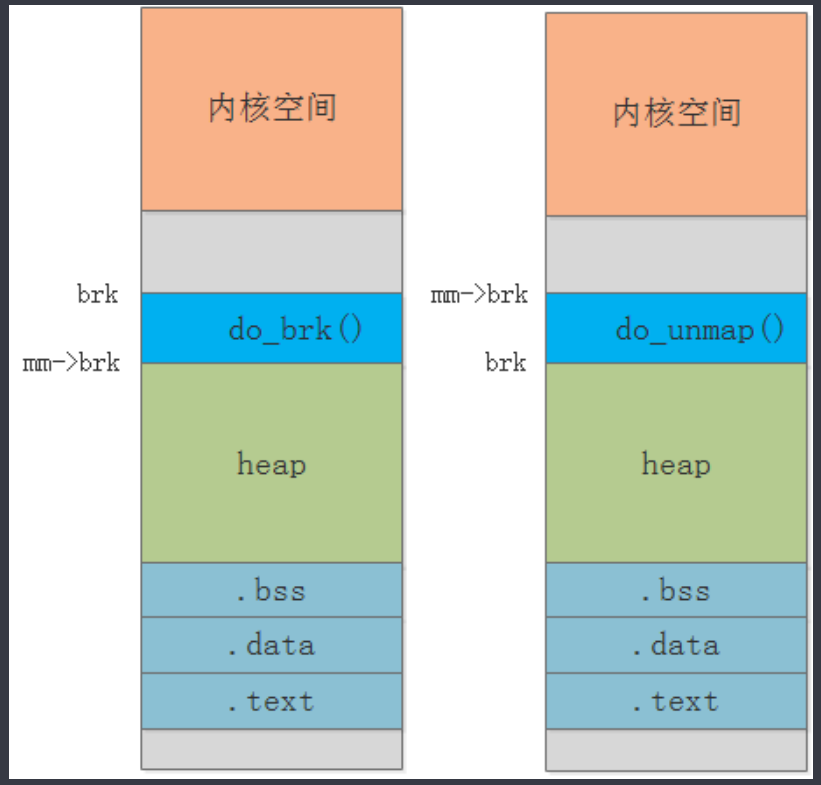
41 私有映射和共享映射

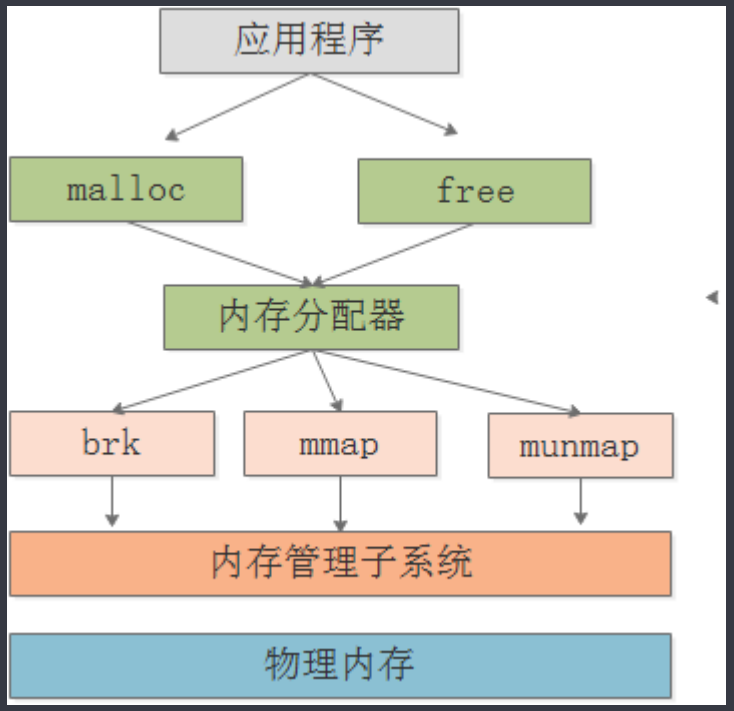
- 映射组合及应用场合
  - 文件共享映射
  - 文件私有映射
  - 匿名共享映射
  - 匿名私有映射



42 系统调用brk实现机制

- 系统调用：brk
  - 用在什么地方？
  - 在什么时候触发brk系统调用？
  - brk系统调用过程分析
  - 思考：为什么堆和栈之间的大片“空间”不能用？

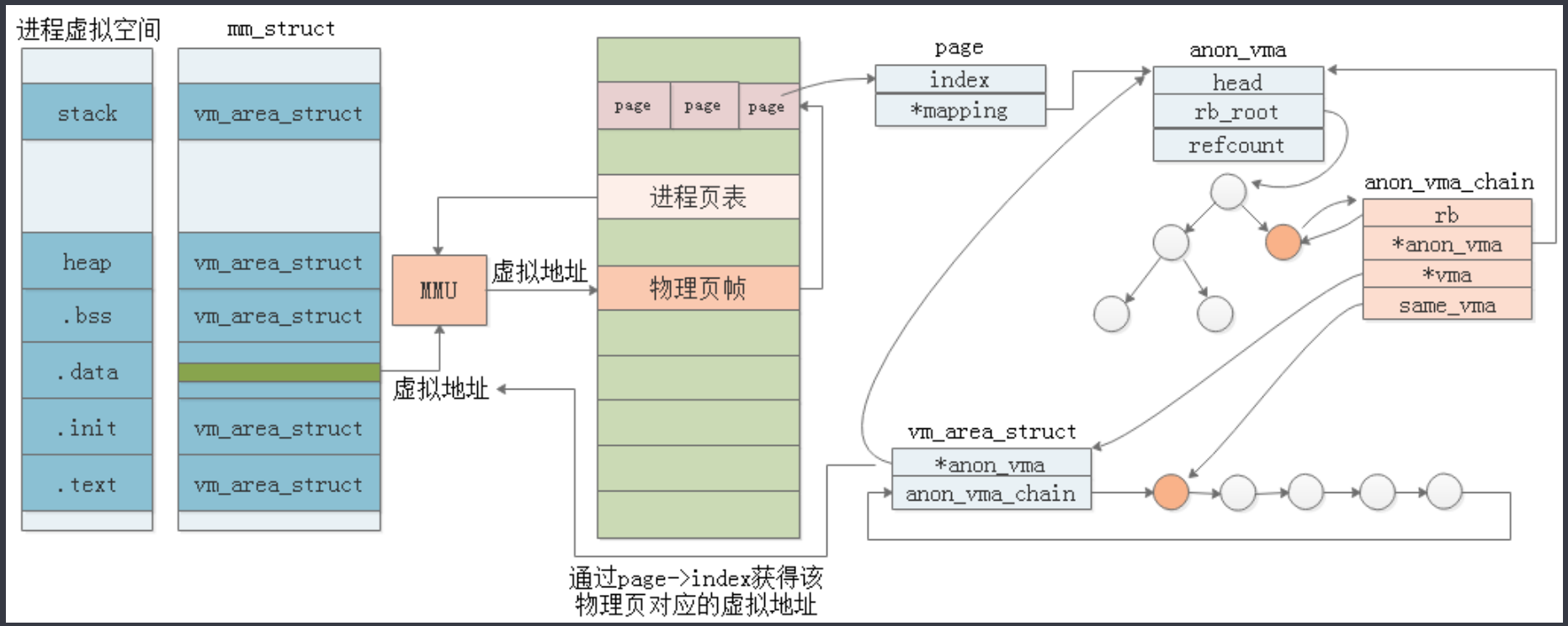




43 反向映射

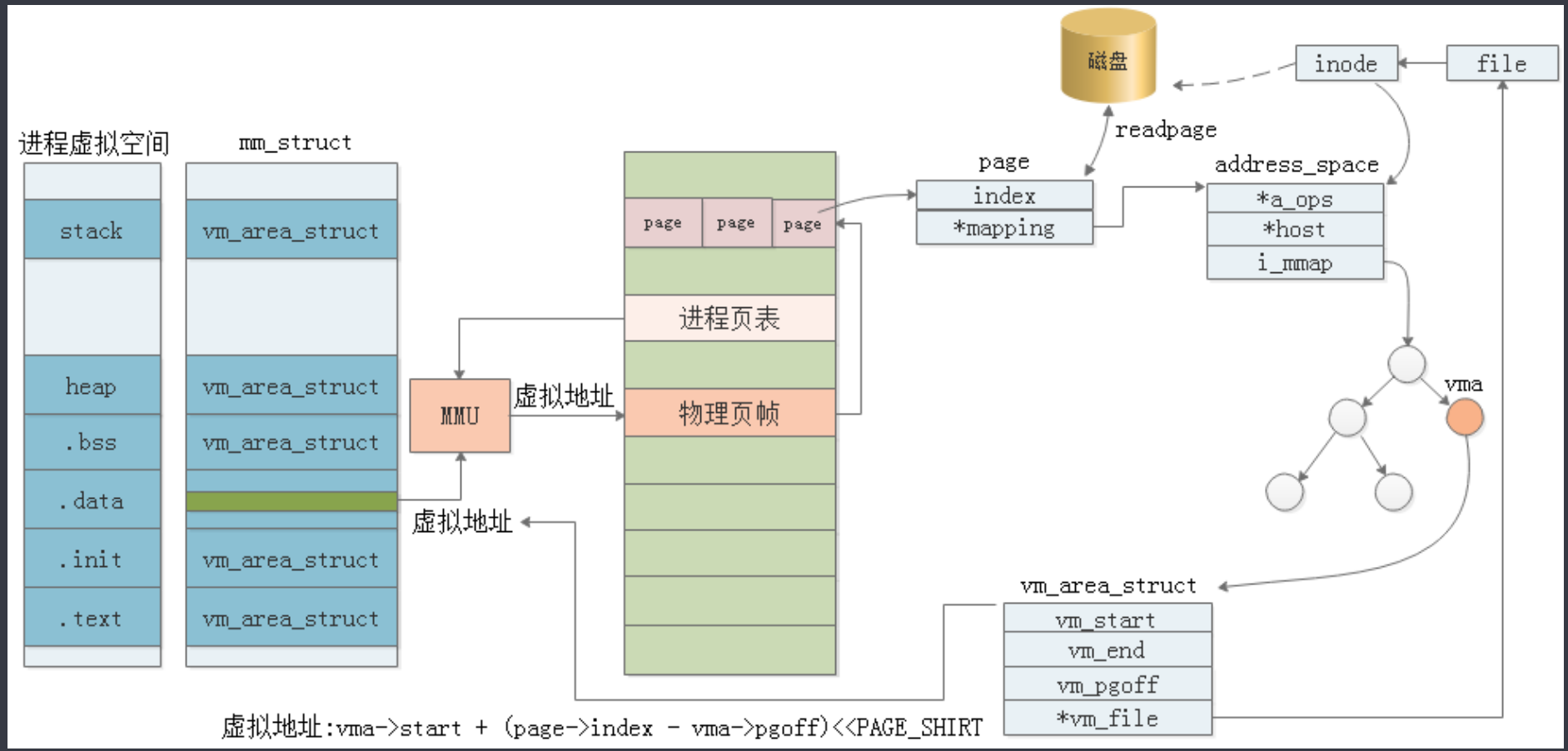
- 什么是反向映射？
  - 反向映射与正向映射
  - 反向映射的应用场景
    - 内存回收、页面迁移
    - 碎片整理、CMA回收、巨型页
  - 匿名页反向映射
  - 文件页反向映射

- 匿名页的反向映射
  - 核心结构体：anon_vma、anon_vma_chain
  - 核心结构体：page、vm_area_struct



• 文件页的反向映射

- 核心结构体：page、address_space
- 核心结构体：vm_area_struct



44 小结

- 小结
  - 内存管理框架回顾(内存管理框架图.pdf)
  - 核心知识点
    - 物理内存管理：memblock、伙伴系统、slab
    - 虚拟内存管理：线性映射、高端内存映射
    - MMU：TLB、Table walk unit
    - 页表：临时页表、二级页表、内核页表、用户页表
    - 映射：mmap、ioremap、反向映射

- 未来...
  - 内存管理框架图.pdf
  - 博客：内存申请与释放接口大全
  - 完善、勘误