Memory Management

课程内容

日期	知识模块	知识点
2月20日	课程介绍	Linux 内核基本结构、Linux的历史、开源基础知识简介、驱动程序介绍
2月27日	实验课一	内核编译，内核补丁
3月5日	内核编程基础知识概述	内核调试技术、模块编程、源代码阅读工具、linux启动过程、git简介
3月12日	实验课2	内核调试
3月19日 3月26日	进程管理与调度	Linux进程基本概念、进程的生命周期、进程上下文切换、Linux 进程调度策略、调度算法、调度相关的调用
4月2日	实验课3	提取进程信息
4月9日	系统调用、中断处理	系统调用内核支持机制、系统调用实现、Linux中断处理、下半部
4月16日	实验课4	添加系统调用、显示系统缺页次数
4月23日	内核同步	原子操作、自旋锁、RCU、内存屏障等linux内核同步机制
4月30日	内存管理	内存寻址、Linux页式管理、物理页分配伙伴系统、Slab管理、进程地址空间
5月14日	实验课5	观察内存映射、逻辑地址与物理地址的对应
5月21日	文件系统	Linux虚拟文件系统、Ext2/Ext3/Ext4文件系统结构与特性
5月28日	Linux设备驱动基础字符设备驱动程序设计	Linux设备驱动基础、字符设备创建和加载、字符设备的操作、对字符设备进行poll 和select的实现、字符设备访问控制、IOCTL、阻塞IO、异步事件等
6月4日	基于linux的容器平台技术概述	虚拟化技术与容器、Docker概述、Kubernetes概述
	实验课6	Docker对容器的资源限制
6月11日	报告课	期末课程报告

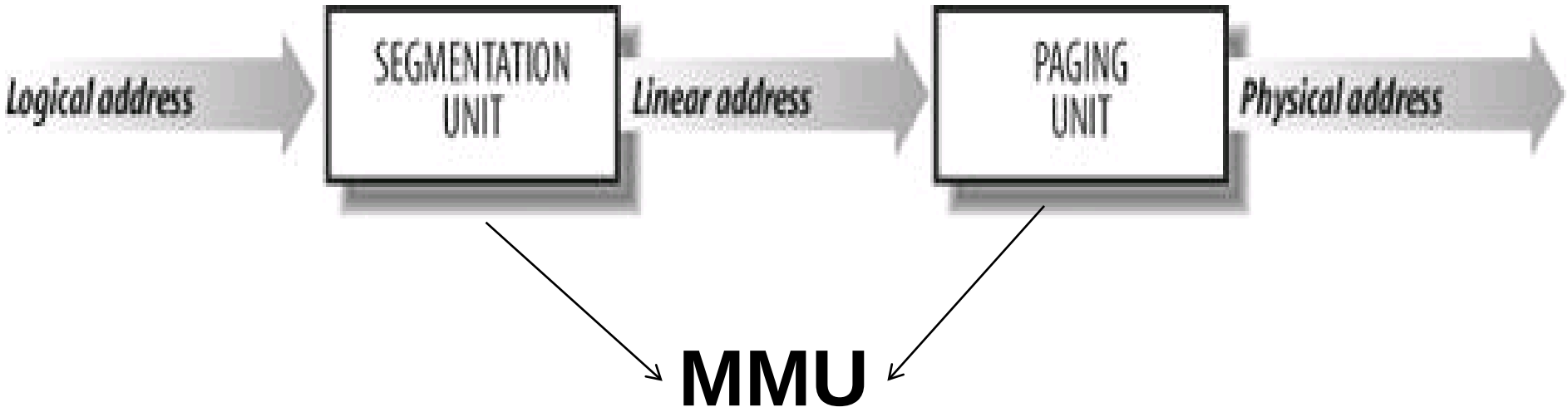
Agenda

1. **Memory Addressing**
2. **Introduction to Linux Physical and Virtual Memory**
3. **Allocators**
   a) **vmalloc(Noncontiguous memory area management)**
   b) **Physical Page Allocation**
   c) **sla/ub**
4. **Process address space**

X86 系列的三种类型地址

- 逻辑地址(Logical address)
  - 机器语言指令中，用于指定一个操作数或一条指令的地址;
  - 包括段(segment) 和偏移量(offset)(偏移量是从段的起始位置开始计算的)
- 线性地址(Linear address) (也称virtual address)
  - 32-bit unsigned integer 的地址寻址空间是 4 GB ， 也就是 4,294,967,296
  - 从0x00000000 到 0xffffffff
- 物理地址
  - 最终用来访问物理内存单元的地址

三种类型地址之间的关系



保护模式中的段和寻址过程

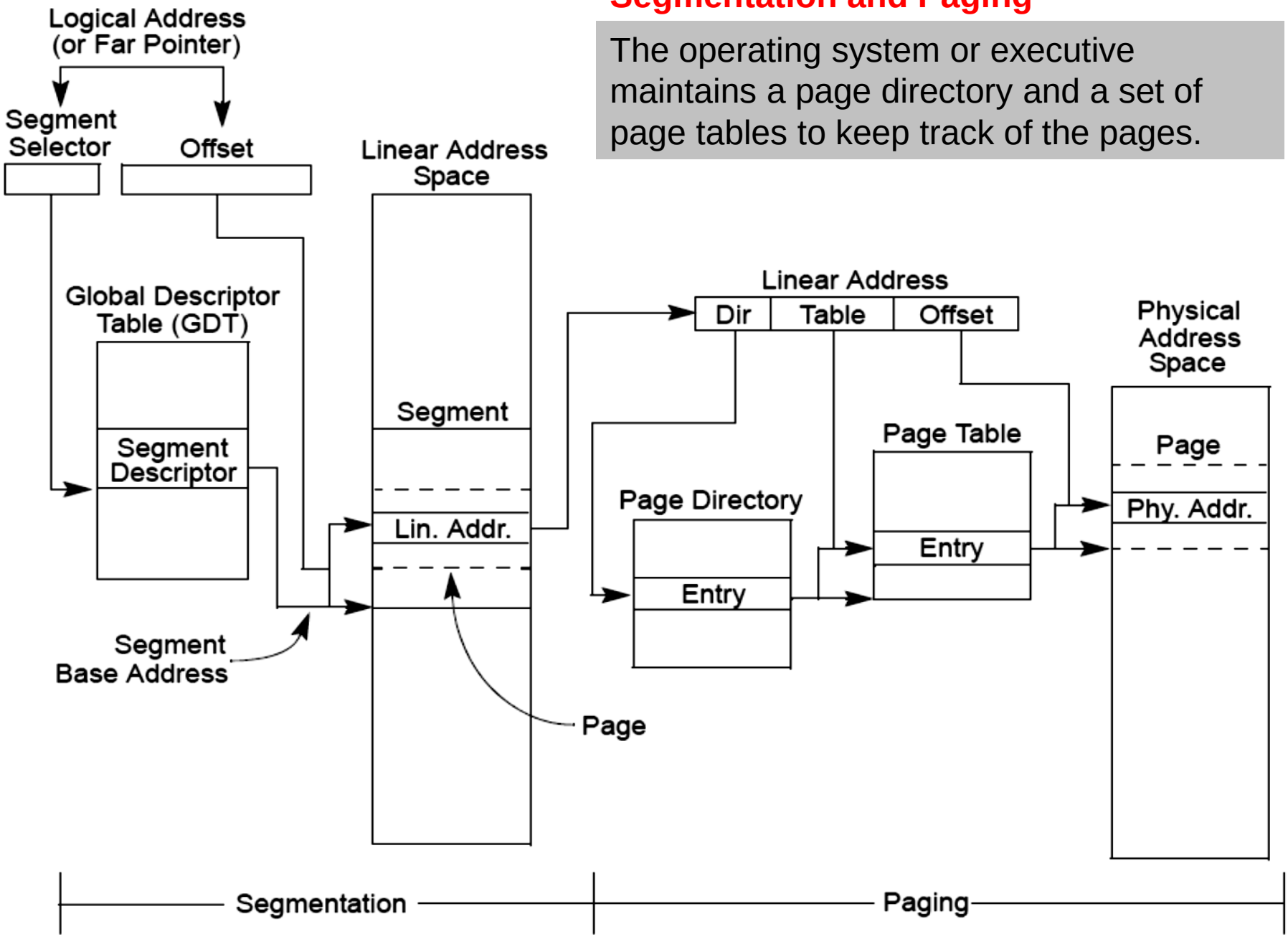
IA-32 体系结构的内存管理设备（**facilities**）被划分成两部分：分段（**segmentation**） 和分页（ **paging**）.

- 分段（**segmentation**）提供了一种分离代码、数据和堆栈模块的机制，从而使多个进程或任务能够运行在同一个处理器上而互不影响。即负责将逻辑地址转换为线性地址。
- 分页（**paging**）提供了一种实现传统的需求分页（**demand-paged**），虚拟存储器机制，从而在程序运行时可将所需要的页映射到物理内存。不过内存分页则是可选的。即将线性地址转换为真正的物理地址。

Linux同时采用了这两种管理机制。

Segmentation and Paging

The operating system or executive maintains a page directory and a set of page tables to keep track of the pages.



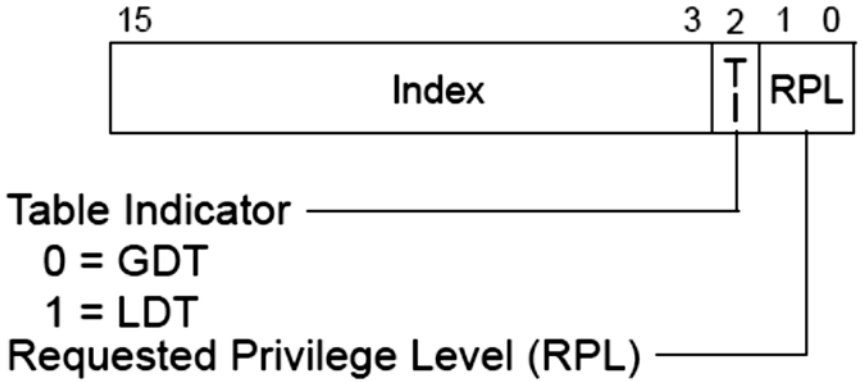
Agenda

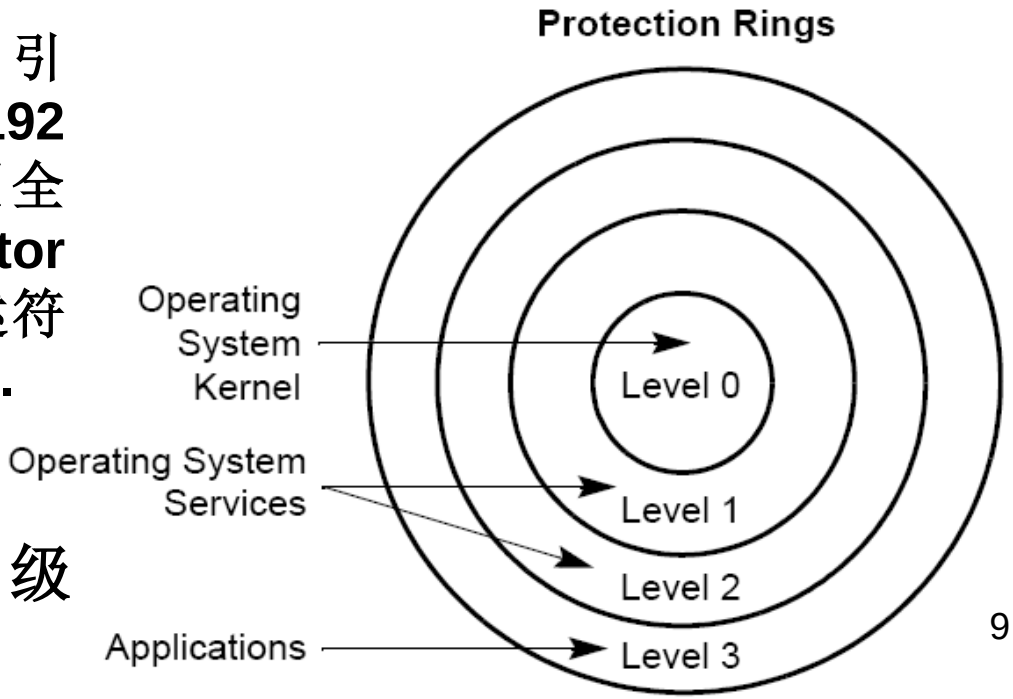
1. **Memory Addressing**
   a) **Segmentation**
   b) **Page Table Management**
2. **Introduction to Linux Physical and Virtual Memory**
3. **Allocators**
   a) **vmalloc(Noncontiguous memory area management)**
   b) **Physical Page Allocation**
   c) **sla/ub**
4. **Process address space**

Segment Selectors

Segment Selector (段标志符，也称段选择子)，对应一个16位的域；常见的格式为：

- **第0位和第1位对应请求优先级别 (RPL, Request Privilege Level)**
- **第 2 位 对 应 表 指 示 器 (Table Indicator), GDT or LDT**
- **第 3 位 到 第 15 位 对 应 索 引 index — Selects one of 8192 descriptors in the GDT（全局描述符表Global Descriptor Table） or LDT（局部描述符表Local Descriptor Table）.**





RPL: Linux只用 0级 和 3 级

RPL: (Request Privilege Level)访问请求的特权级别。RPL保存在选择子的最低两位。RPL说明的是进程对段访问的请求权限，意思是当前进程想要的请求权限。RPL的值由程序员自己来自由的设置，可以看成是每次访问时的附加限制

CPL：(Current Privilege Level)当前进程的执行特权级别，是当前正在执行的代码所在段的特权级，在CS寄存器的低两位。

DPL：(Descriptor Privilege Level)要访问的内存所要求的特权级别。DPL存储在段描述符中，规定访问该段的权限级别(Descriptor Privilege Level)，每个段的DPL固定。

- 访问数据段、门描述符时，max (CPL , RPL) <= DPL 时表示允许访问
- 访问堆栈段要求CPL =RPL= DPL
- 访问代码段根据一致代码段和非一致代码段(前者>=,后者=)

The Linux GDT

- **GDT是一个全局数据结构，系统中的所有程序和任务都可以使用 GDT 描述符表， 因此 GDT 中包含着系统中所有任务都可用的那些段描述符， 即公用的段描述符。**
- **可以出现在 GDT 中的描述符有： 代码段、 数据段、 TSS、LDT 和调用门、 任务门描述符。 中断门、 陷阱门不能出现在GDT 中。GDT 中的第一个描述符保留不用（全为 0）**
- **GDT：单处理器，有一个；多处理器，对每个处理器都有一个**
- **GDTR：GDT的基址、界限、属性等**
- **LGDT & SGDT**
- **初始化：setup_gdt()**

/arch/x86/include/asm/desc.h

```
struct gdt_page {
        struct desc_struct gdt[GDT_ENTRIES];
} __attribute__((aligned(PAGE_SIZE)));

DECLARE_PER_CPU_PAGE_ALIGNED(struct
gdt_page, gdt_page);
```

The Linux LDT

- 局部描述符表（**LDT**） 是一个段， 存放局部的、不需要全局共享的段描述符
- 系统中可以定义多个局部描述符表（**LDT**）。每一个 **LDT**在 **GDT** 中都有一个段描述符。
- **linux**用户态下程序一般不使用**LDT**。
- 缺省**LDT**：所有进程共享，**GDT**中有一个包括缺省**LDT**段的段描述符（进程可以创建自己的**LDT**）
- **LDTR**：当前任务的**LDT**的基址、界限、属性等
- **LLDT & SLDT**

segment registers

- 程序要访问段，必须把段的 **segment selector** 加载到段寄存器 **segment registers** 中。尽管 **segment selector** 有很多，但是只有**6**个 **segment registers**，即**CS(Code Segment)**、**SS(Stack Segment)**、**DS(Date Segment)**、**ES(Extra Segment)**、**FS**、**GS**。
- 段寄存器包括可见部分和隐藏部分**(**当一个段选择子被装入到一个段寄存器的可见部分的时候，处理器根据该选择子找到对应的段描述符，然后将其中的基地址、段界限、存取控制信息等装入到段寄存器的隐含部分。**)**

Visible Part	Hidden Part	
Segment Selector	Base Address, Limit, Access Information	CS
		SS
		DS
		ES
		FS
		GS

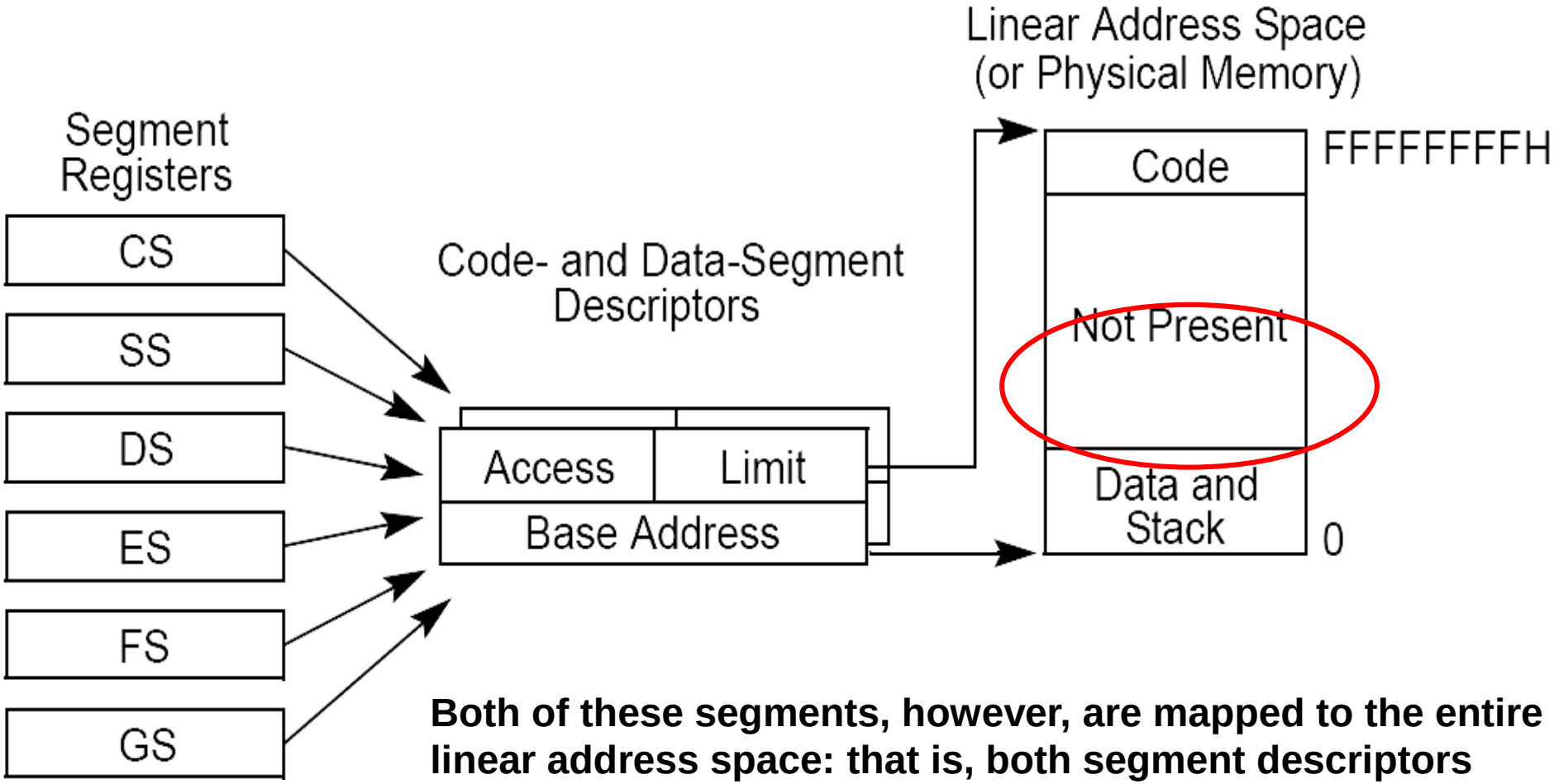
三种使用segment的模式（Intel）

- **Basic Flat Model**
- **Protected Flat Model**
- **Multi-Segment Model**

Basic Flat Model

- 基本的平展模型：最简单，操作系统和应用程序可以访问一个连续的、未分段的地址空间。
- 在**IA-32**体系结构中实现**basic flat model,**至少需要创建代码段和数据段两个段描述符，都映射到整个线性地址空间，有同样的基地址**0**以及段限制 **4GB**。
- 对系统程序员和应用程序员隐藏了分段机制。
- **By setting the segment limit to 4GBytes, the segmentation mechanism is kept from generating exceptions for out of limit memory references, even if no physical memory resides at a particular address**

Basic Flat Model

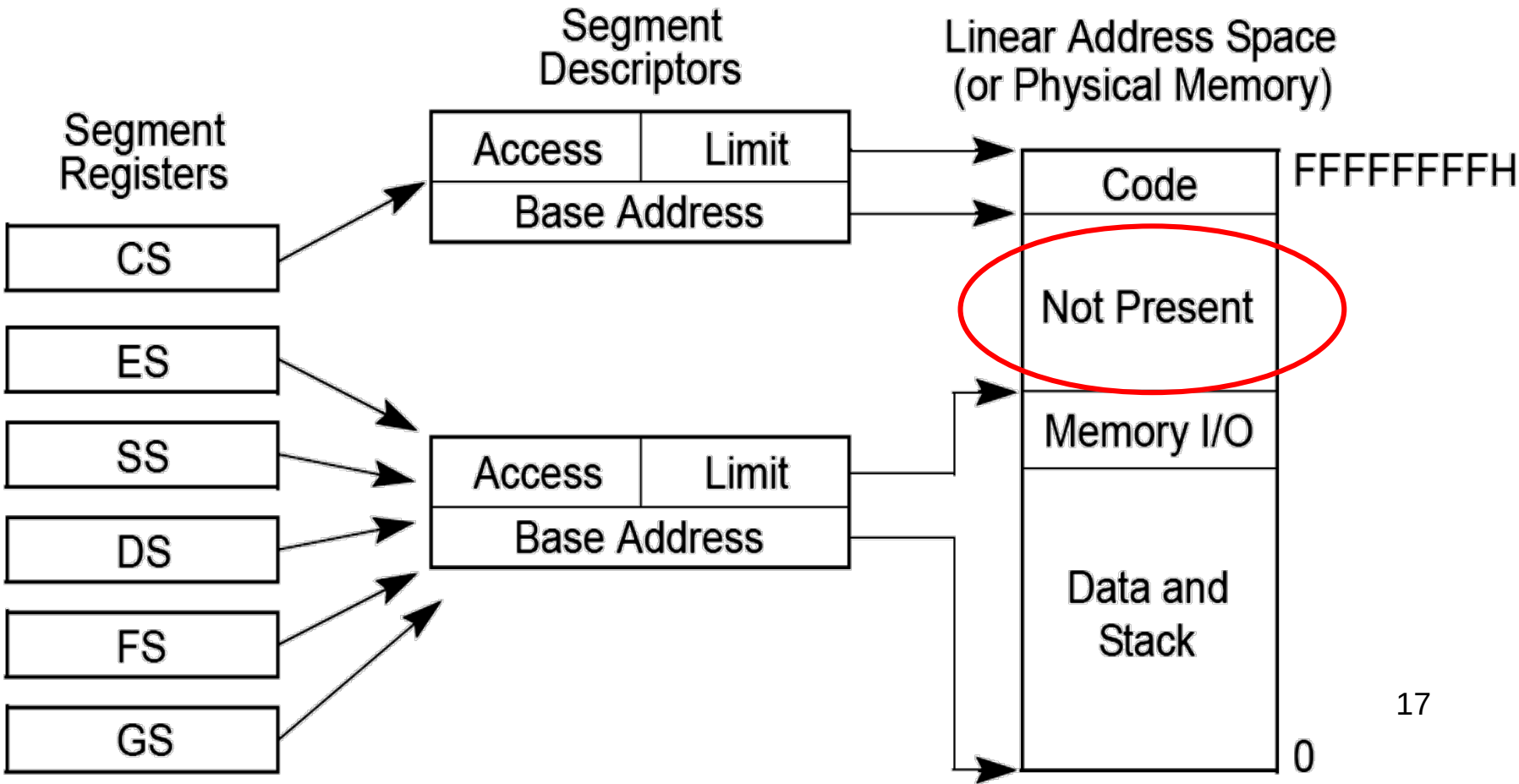


Both of these segments, however, are mapped to the entire linear address space: that is, both segment descriptors have the same base address value of 0 and the same segment limit of 4 GBytes.

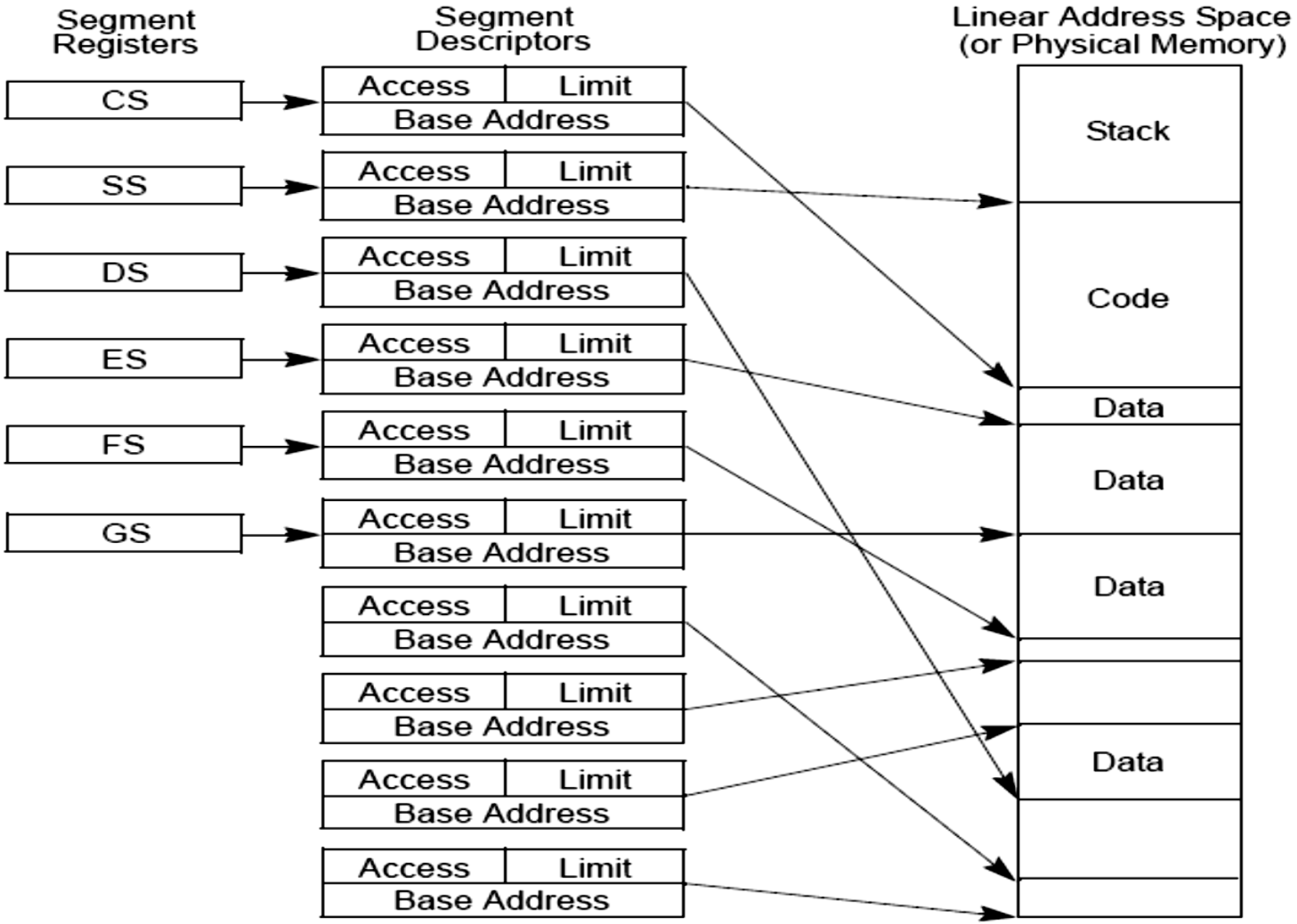
Linux 采用basic flat model

Protected Flat Model

- 类似Basic Flat Model，只是段限制(segment limits) 设置仅仅包括物理内存存在的范围。
- 访问不存在的物理内存地址会产生异常。



Multi-Segment Model



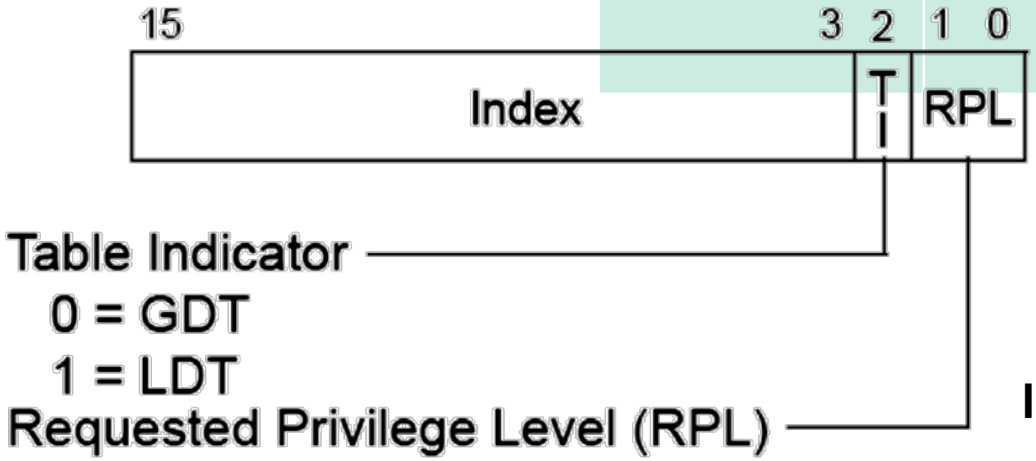
Segmentation in Linux

- **Linux uses segmentation only when required by the 80x86 architecture.**
- 所有运行于用户态的 **Linux** 进程在用户模式下使用同样的一对段： **user code segment** 和 **user data segment.** 同样在内核模式下的所有进程使用同样的段称为**kernel code segment** 和 **kernel data segment**。
- 分别定义为： **__USER_CS, __USER_DS, __KERNEL_CS,** 和 **__KERNEL_DS,** 例如访问内核代码段，内核只是加载由 **__KERNEL_CS** 宏得到的值到 **CS** 段寄存器中。
- **linux/arch/x86/include/asm/segment.h**

```
#define GDT_ENTRY_KERNEL_CS             12
#define GDT_ENTRY_KERNEL_DS             13
#define GDT_ENTRY_DEFAULT_USER_CS       14
#define GDT_ENTRY_DEFAULT_USER_DS       15
```

```
#define __KERNEL_CS             (GDT_ENTRY_KERNEL_CS*8)
#define __KERNEL_DS             (GDT_ENTRY_KERNEL_DS*8)
#define __USER_DS               (GDT_ENTRY_DEFAULT_USER_DS*8 + 3)
#define __USER_CS               (GDT_ENTRY_DEFAULT_USER_CS*8 + 3)
```

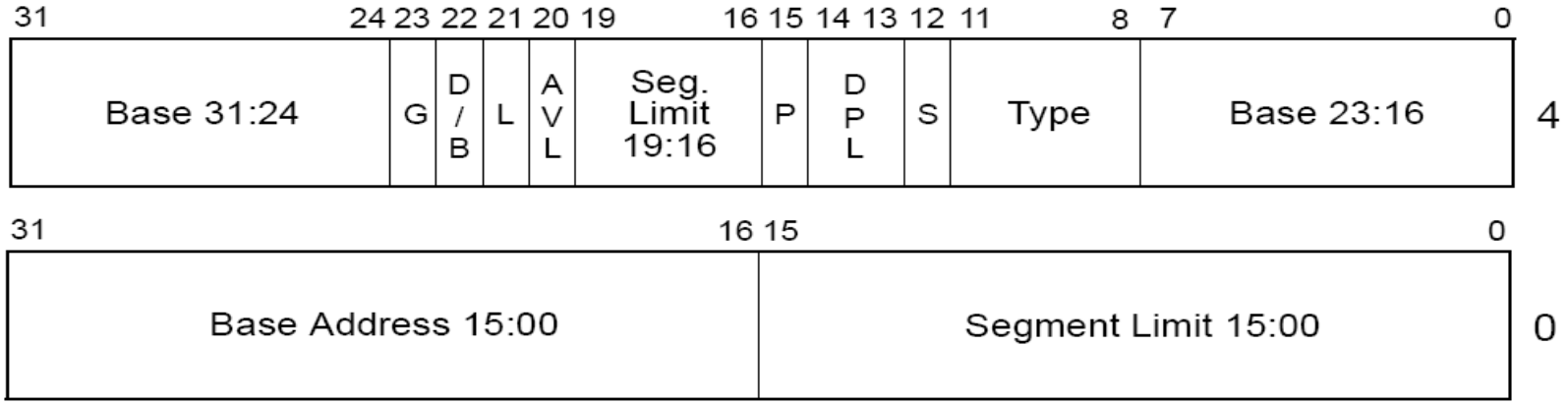
KERNEL CS	KERNEL DS	USER CS	USER DS
1100	1101	1110	1111
1100000	1101000	1110000	1111000
		1110011	1111011



linux/arch/x86/include/asm/segment.h

Segment Descriptors

段描述符（**Segment Descriptors**）是GDT或LDT表中的一个数据结构项，它提供了段的大小、位置、控制权限和状态信息等等。段描述符通常由编译器、链接器、加载器或操作系统创建，而不是应用程序创建。



L — 64-bit code segment (IA-32e mode only)
AVL — Available for use by system software
BASE — Segment base address
D/B — Default operation size (0 = 16-bit segment; 1 = 32-bit segment)
DPL — Descriptor privilege level
G — Granularity
LIMIT — Segment Limit
P — Segment present
S — Descriptor type (0 = system; 1 = code or data)
TYPE — Segment type

/arch/x86/include/asm/desc_defs.h

```
struct desc_struct {
        u16  limit0;
        u16  base0;
        u16  base1: 8, type: 4,
                s: 1, dpl: 2, p: 1;
        u16  limit1: 4, avl: 1, l: 1,
                d: 1, g: 1, base2: 8;
} __attribute__((packed));
```

Values of the Segment Descriptor fields

Segment	Base	G	Limit	S	Type	DPL	D/B	P
user code	0x00000000	1	0xfffff	1	10	3	1	1
user data	0x00000000	1	0xfffff	1	2	3	1	1
kernel code	0x00000000	1	0xfffff	1	10	0	1	1
kernel data	0x00000000	1	0xfffff	1	2	0	1	1

- 对 KERNEL_CS：DPL=0，表示 0 级；S 位为 1，表示代码段或数据段：type 为 1010，表示代码段，可读，可执行，尚未受到访问。
- 对 KERNEL_DS：DPL=0，表示 0 级；S 位为 1，表示代码段或数据段：type 为 0010 表示数据段，可读，可写，尚未受到访问。
- 对 USER_CS：DPL=3，表示 3 级：S 位为 1，表示代码段或数据段；type 为 1010，表示代码段，可读，可执行，尚未受到访问。
- 对 USER_DS：即下标为 5 时，DPL=3，表示 3 级：S 位为 1，表示代码段或数据段，type 为 0010，表示数据段，可读，可写，尚未受到访问。

```
47enum {
48      DESC_TSS = 0x9,
49      DESC_LDT = 0x2,
50      DESCTYPE_S = 0x10,  //!system
51};
```

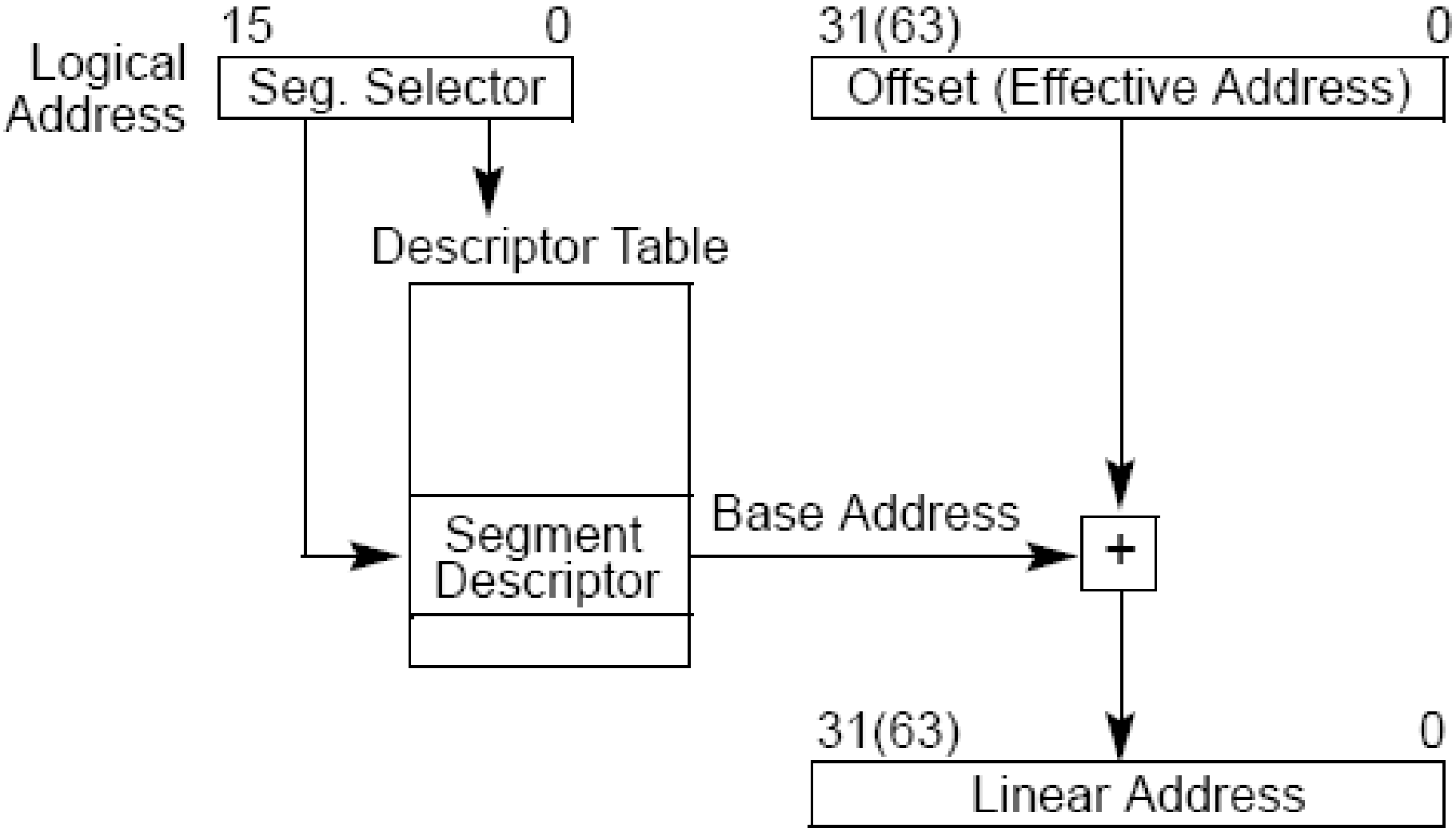
linux/arch/x86/include/asm/desc_defs.h

/arch/x86/kernel/head_32.S

```
.quad 0x00cf9a000000ffff
.quad 0x00cf92000000ffff
```

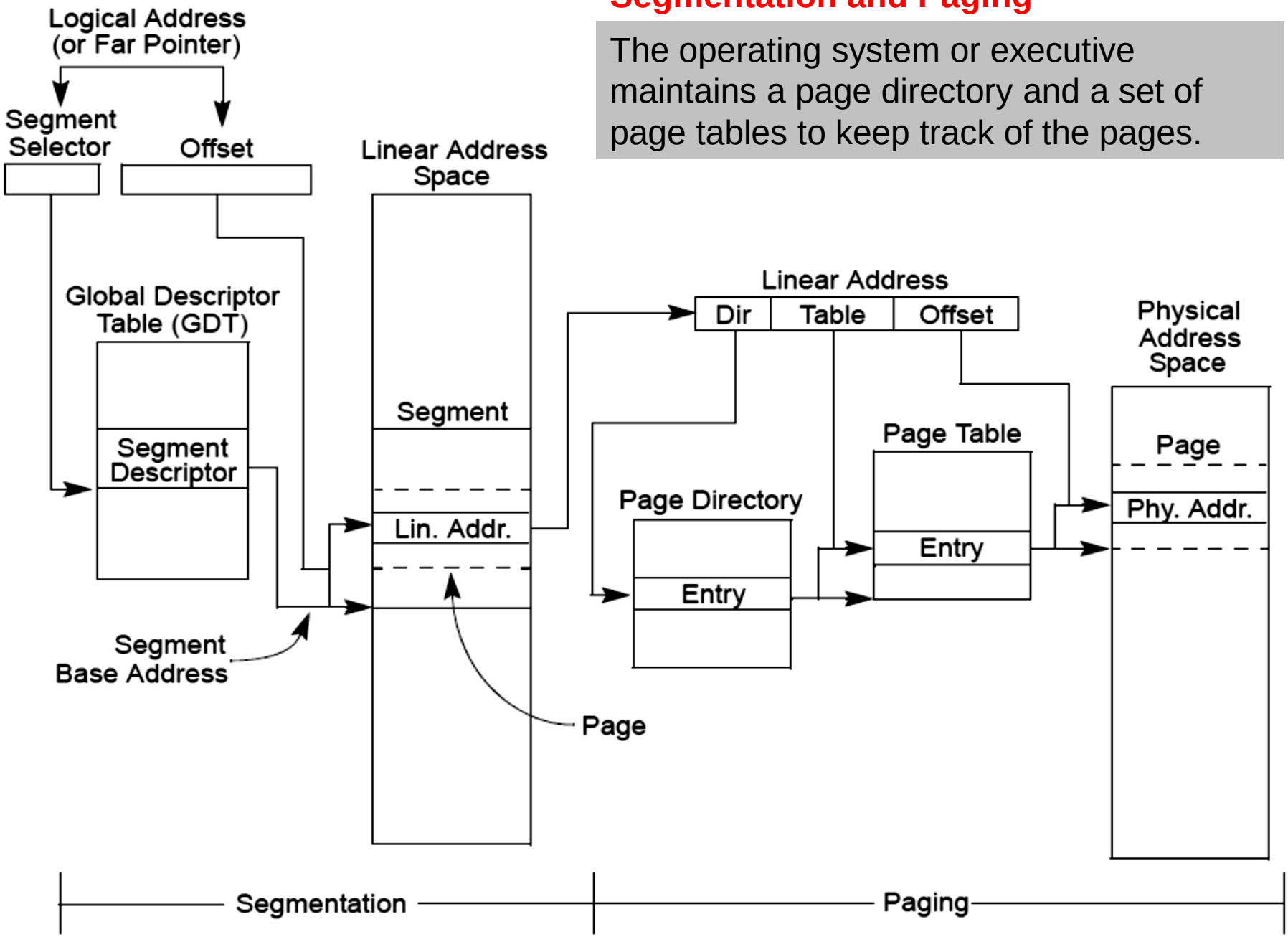
```
struct desc_struct {
        u16  limit0;
        u16  base0;
        u16  base1: 8, type: 4,
             s: 1, dpl: 2, p: 1;
        u16  limit1: 4, avl: 1, l: 1,
             d: 1, g: 1, base2: 8;
} __attribute__((packed));
```

逻辑地址和线性地址的翻译



Segmentation and Paging

The operating system or executive maintains a page directory and a set of page tables to keep track of the pages.



Agenda

1. **Memory Addressing**
   a) **Segmentation**
   b) **Page Table Management**
2. **Introduction to Linux Physical and Virtual Memory**
3. **Allocators**
   a) **vmalloc(Noncontiguous memory area management)**
   b) **Physical Page Allocation**
   c) **sla/ub**
4. **Process address space**

Page and page frame

- Page: **a block of data**, which may be stored in any page frame or on disk.

- Page frame: **physical addresses in main memory.**

- Because it is 12 bits long, each page consists of 4096 bytes of data. Because each page frame has a 4KB capacity, its **physical address must be a multiple of 4096.**

- This allows the same page to be stored in a page frame, then saved to disk and later reloaded in a different page frame. This is the basic ingredient of the **virtual memory mechanism.**

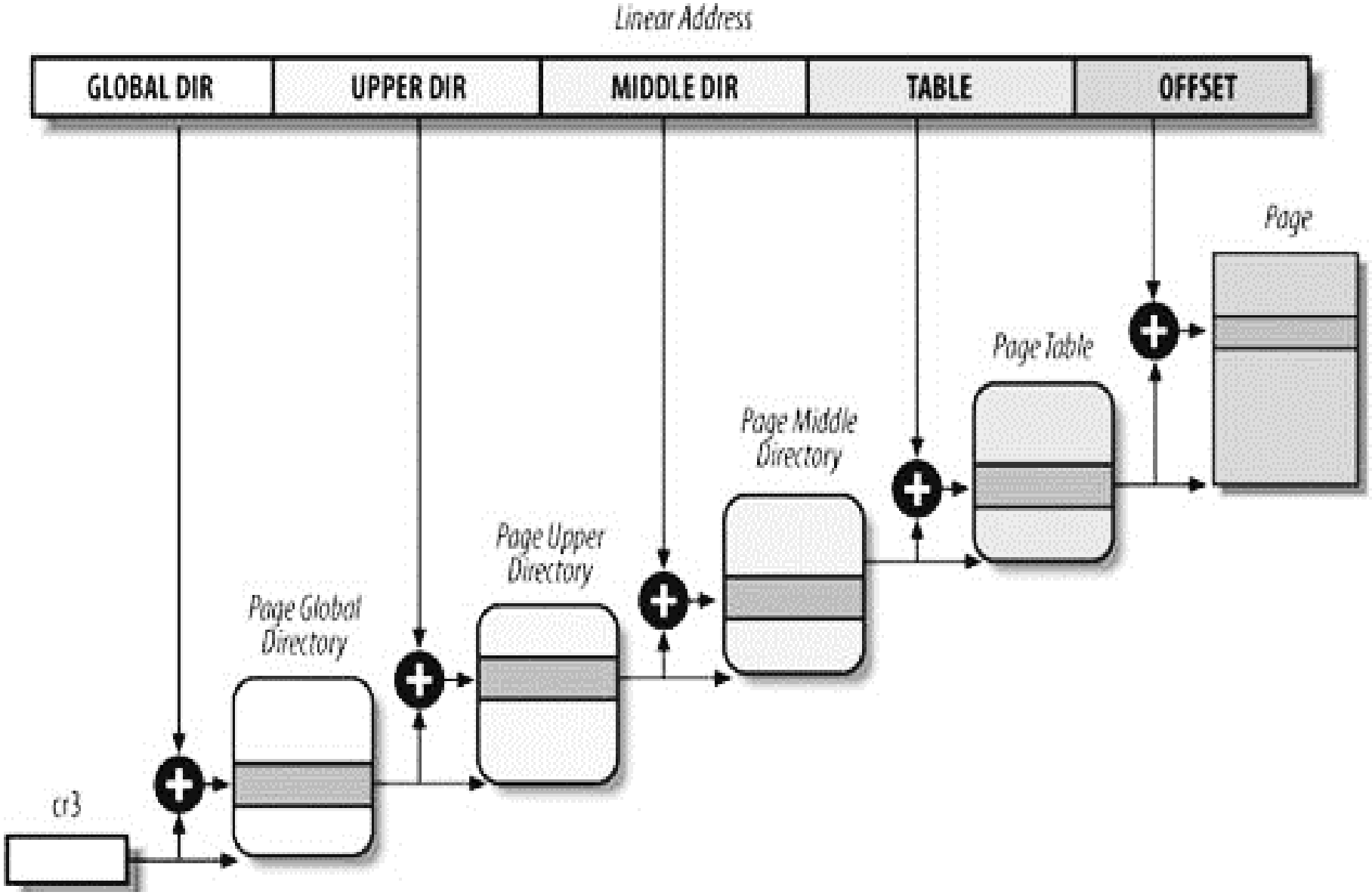
Paging in Linux

Linux从2.6.11版本开始，采用四级分页模型：

- Page Global Directory(PGD) 页全局目录
- Page Upper Directory(PUD) 页上级目录
- Page Middle Directory(PMD) 页中间目录
- Page Table(PT) 页表

Why multi-level?

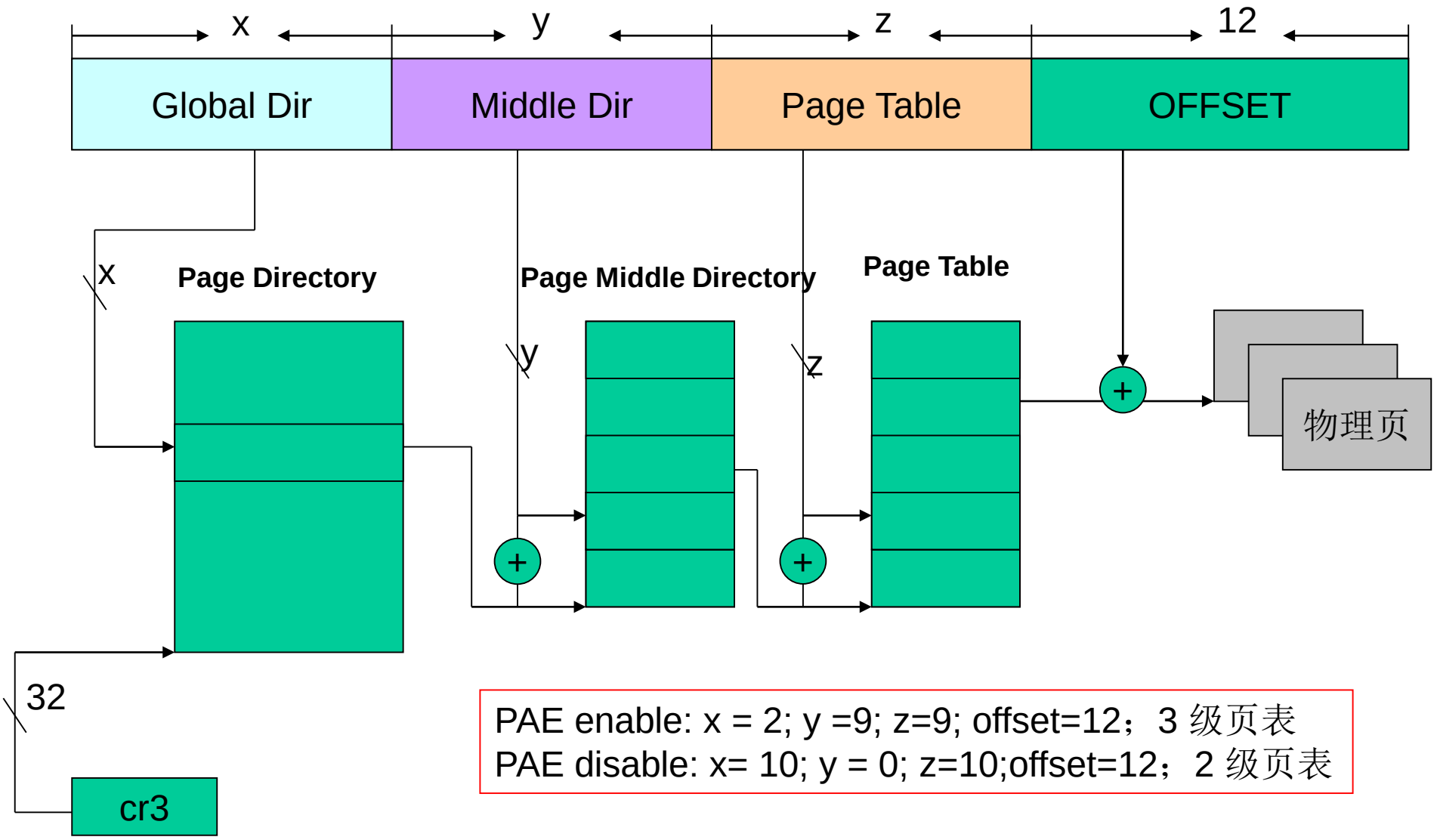
The Linux paging model



Paging Levels

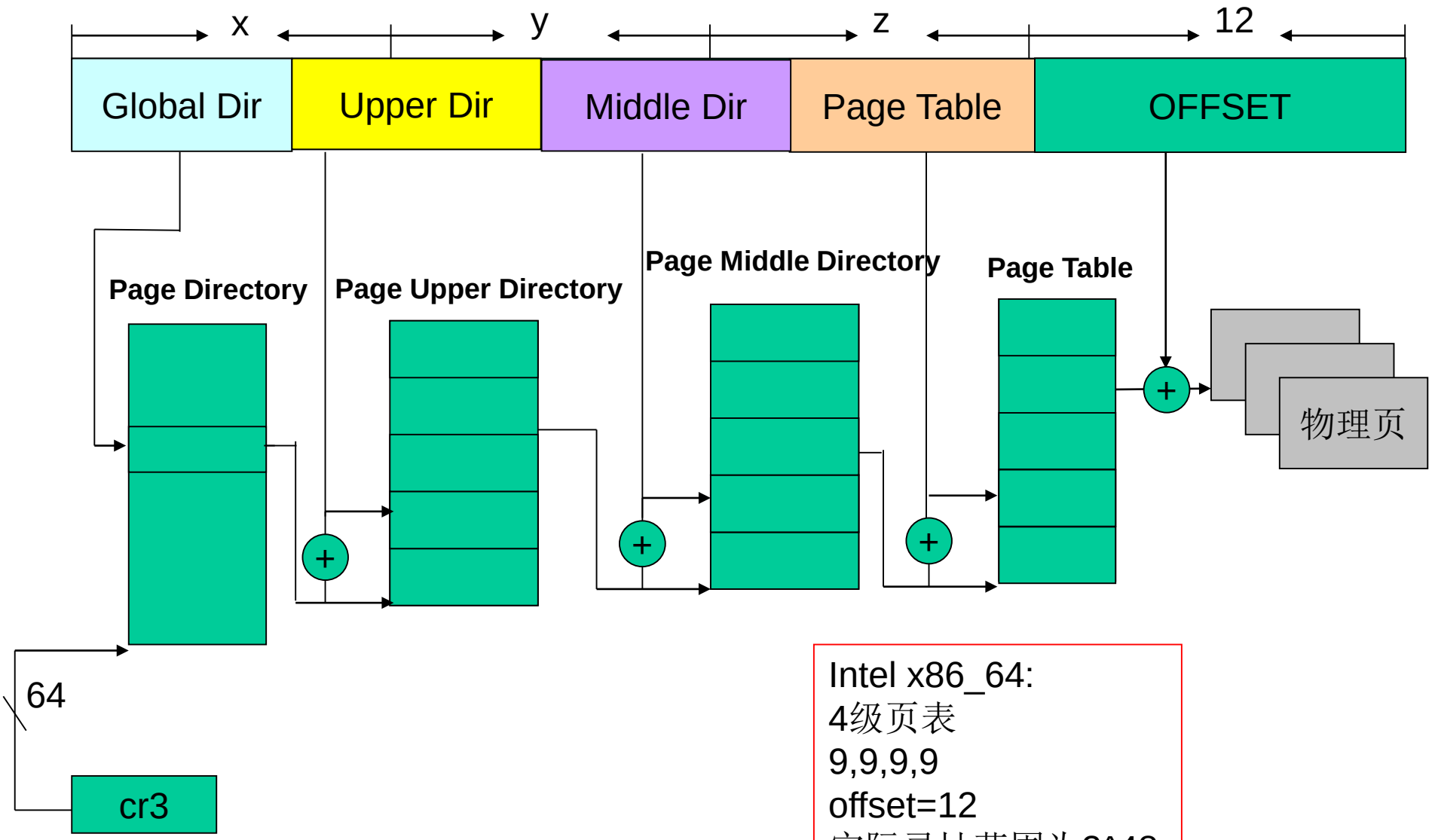
- For **32-bit architectures**, *two paging levels* are sufficient for 32-bit architectures. Linux essentially eliminates the **PUD** and the **PMD** fields by saying that they contain zero bits. The kernel keeps a position for the **PUD** and the **PMD** by setting the number of entries in them to 1 and mapping these two entries into the proper entry of the **PGD**.
- For **32-bit architectures with the PAE** enabled, *three paging levels* are used. The Linux's **PGD** corresponds to the 80 x 86's **PDPT**, the **PUD** is eliminated, the **PMD** corresponds to the 80 x 86's Page Directory, and the Linux's Page Table corresponds to the 80 x 86's Page Table.
- **64-bit architectures** require a higher number of paging levels. Up to version 3.0.1, the Linux paging model consisted of **three paging levels**. Starting with version 3.0.1, a **four-level paging** model has been adopted, the size of each part depends on the computer architecture.
  - for **64-bit architectures** three or four levels of paging are used depending on **the linear address bit splitting performed by the hardware**.

32-bit 线性地址时候的三级或二级页表结构的时候，没有PUD



PAE enable: x = 2; y =9; z=9; offset=12；3 级页表
PAE disable: x= 10; y = 0; z=10;offset=12；2 级页表

64bit 线性地址时候的页表结构



Intel x86_64:
4级页表
9,9,9,9
offset=12
实际寻址范围为2^48

Page Size Extension (PSE)

- Starting with the Pentium model, 80x86 microprocessors introduce extended paging.
- allows page frames to be **4MB** instead of 4KB in size. in these cases, the kernel **can do without intermediate Page Tables** and thus save memory and preserve TLB(Translation lookaside buffer) entries.
- the paging unit divides the 32 bits of a linear address into two fields:
  - **Directory**: The most significant 10 bits.
  - **Offset**: The remaining 22 bits.
- enabled by setting the **PSE** flag of **cr4.**
- **PSE-36: 14+22,** address up to 2^{36} = **64G**B of RAM

TLB和CPU里的一级、二级缓存之间不存在本质的区别，前者缓存页表数据，而后两个缓存实际数据

Physical Address Extension (PAE)

- Starting with the Pentium Pro, all Intel processors are now able to address up to 2^{36} = 64 GB of RAM.
- translates 32-bit linear addresses into 36-bit physical ones.
- a PAE Page Table entry: 36 =12 flag bits + 24 physical address bits
  - so the Page Table entry size has been doubled from 32 bits to 64 bits. As a result, a 4KB PAE Page Table includes 512 entries instead of 1,024.
- A new level of Page Table called the **Page Directory Pointer Table (PDPT)** consisting of four 64-bit entries has been introduced (linux中的PGD，由cr3指向其基址，linear address的30-31确定是4个中的哪个).
- activated by setting the Physical Address Extension (PAE) flag in the **cr4.**
- The Page Size (PS) flag in the page directory entry enables large page sizes (2MB when PAE is enabled).

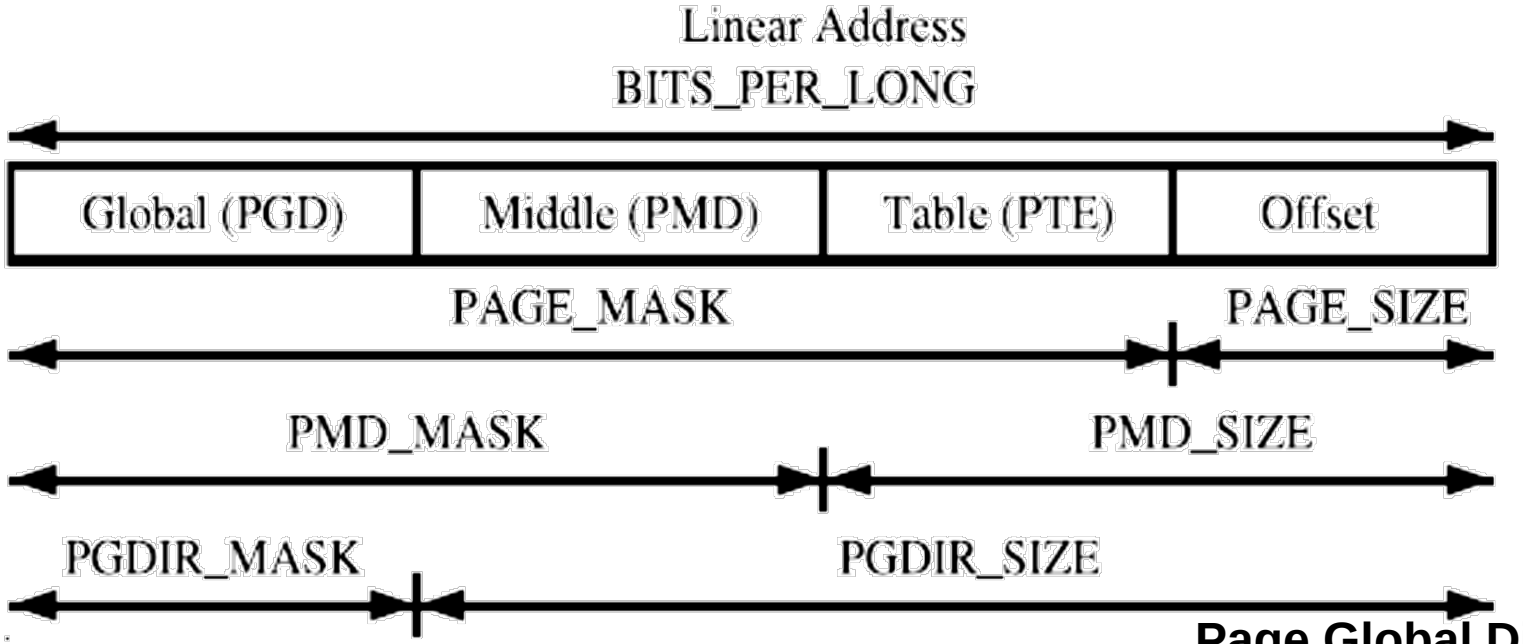
PAE

- When mapping linear addresses to **4KB** pages, the **32** bits of a linear address are interpreted in the following way:
  - cr3: Points to a PDPT.
  - bits 31-30: Point to 1 of 4 possible entries in PDPT;
  - bits 29-21: Point to 1 of 512 possible entries in Page Directory;
  - bits 20-12: Point to 1 of 512 possible entries in Page Table;
  - bits 11-0: Offset of 4-KB page.

- When mapping linear addresses to **2MB** pages, the **32** bits of a linear address are interpreted in the following way:
  - cr3: Points to a PDPT.
  - bits 31-30: Point to 1 of 4 possible entries in PDPT;
  - bits 29-21: Point to 1 of 512 possible entries in Page Directory;
  - bits 20-0: Offset of 2-MB page.

- **The page size of x89-64 ?**

Linear Address Bit Size Macros



Page Global Directory
Page Upper Directory
Page Middle Directory
Page Table Entry

/arch/x86/include/asm/page_types.h

- **#define PAGE_SHIFT 12**
- **#define PAGE_SIZE (_AC(1,UL) << PAGE_SHIFT)**
- **#define PAGE_MASK (~(PAGE_SIZE-1))**

pte_t, pmd_t, pgd_t, pgprot_t定义

{linuxsrcdir}/arch/x86/include/asm/page_types.h

- **typedef struct pgprot { pgprotval_t pgprot; } pgprot_t;//页属性**
- **typedef struct { pgdval_t pgd; } pgd_t;//页目录项**

- **#if PAGETABLE_LEVELS > 3 //x86_64使用4级页表，增加页上级目录**
- **typedef struct { pudval_t pud; } pud_t;**

- **#if PAGETABLE_LEVELS > 2 //开启PAE,使用3级页表，增加页中间目录**
- **typedef struct { pmdval_t pmd; } pmd_t;**

Cont’

{linuxsrcdir}/arch/x86/include/asm/page_64_types.h

- **typedef unsigned long pteval_t;**
- **typedef unsigned long pmdval_t;**
- **typedef unsigned long pudval_t;**
- **typedef unsigned long pgdval_t;**
- **typedef unsigned long pgprotval_t;**

- **typedef struct { pteval_t pte; } pte_t;**

type-conversion macros

- cast an unsigned integer into the required type

#define __pte(x) native_make_pte(x)

__pmd(x)

__pgd(x)

__pgprot(x)

```
static inline pte_t native_make_pte(pteval_t val)
{
        return (pte_t) { .pte = val  };
}
```

- reverse casting

#define pte_val(x) native_pte_val(x)

pmd_val(x)

pgd_val(x)

pgprot_val(x)

```
static inline pteval_t native_pte_val(pte_t pte)
{
        return pte.pte;
}
```

Page table entry flags (pgprot) and Status

- {linuxsrcdir}/arch/x86/include/asm/pgtable_types.h
- #define _PAGE_BIT_PRESENT 0 /* is present */
- #define _PAGE_BIT_RW 1 /* writeable */
- #define _PAGE_BIT_USER 2 /* userspace addressable */
- #define _PAGE_BIT_PWT 3 /* page write through */
- #define _PAGE_BIT_PCD 4 /* page cache disabled */
- #define _PAGE_BIT_ACCESSED 5 /* was accessed (raised by CPU) */
- #define _PAGE_BIT_DIRTY 6 /* was written to (raised by CPU) */
- #define _PAGE_BIT_PSE 7 /* 4 MB (or 2MB) page */
- #define _PAGE_BIT_PAT 7 /* on 4KB pages */
- #define _PAGE_BIT_GLOBAL 8 /* Global TLB entry PPro+ */
- #define _PAGE_BIT_UNUSED1 9 /* available for programmer */
- #define _PAGE_BIT_IOMAP 10 /* flag used to indicate IO mapping */
- #define _PAGE_BIT_HIDDEN 11 /* hidden by kmemcheck */
- #define _PAGE_BIT_PAT_LARGE 12 /* On 2MB or 1GB pages */
- #define _PAGE_BIT_SPECIAL _PAGE_BIT_UNUSED1
- #define _PAGE_BIT_CPA_TEST _PAGE_BIT_UNUSED1
- #define _PAGE_BIT_SPLITTING _PAGE_BIT_UNUSED1 /* only valid on a PSE pmd */
- #define _PAGE_BIT_NX 63 /* No execute: only valid after cpuid check */

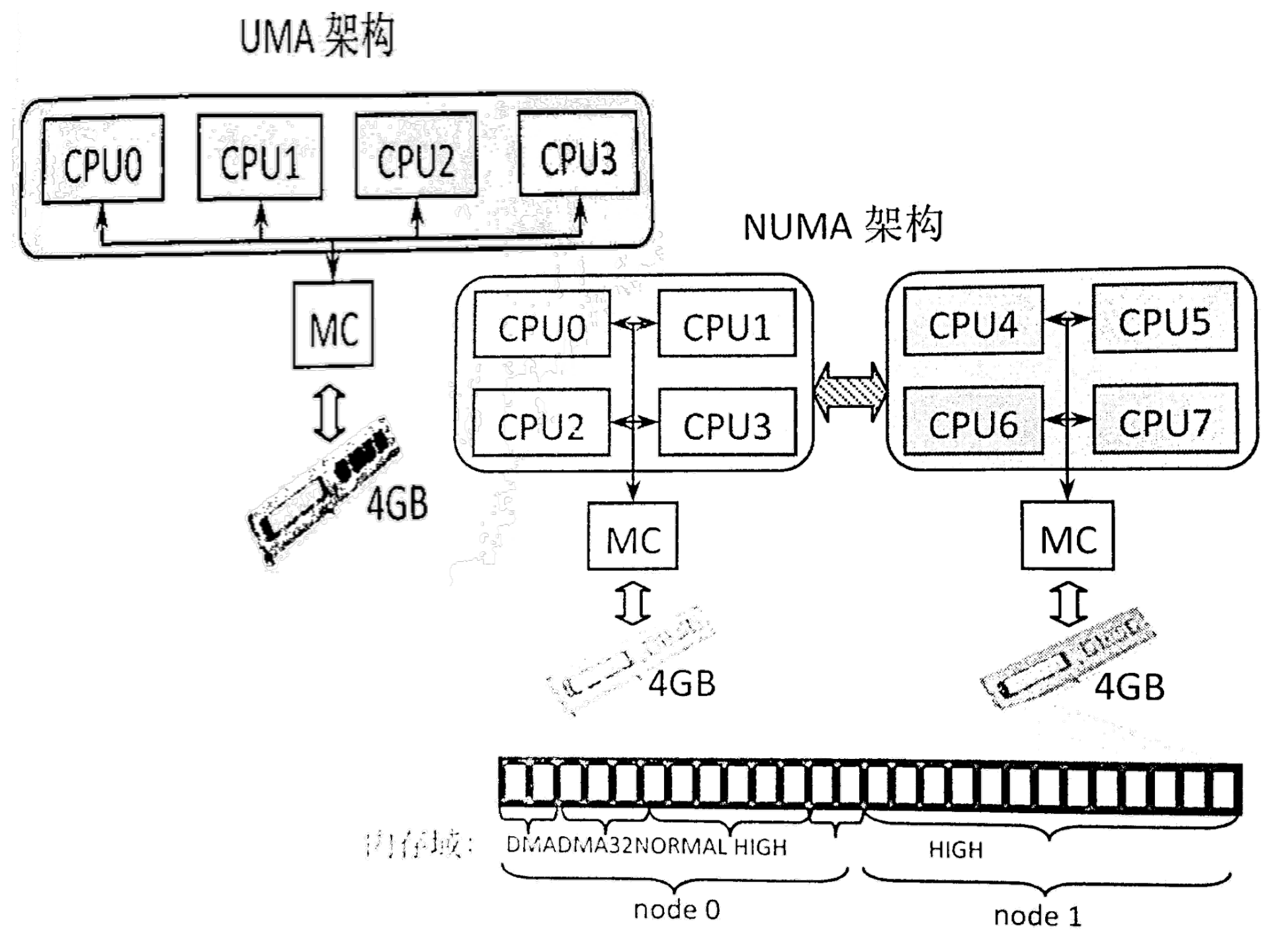
macros and functions to read or modify page table entries

- `pte_none()`, `pmd_none()` and `pgd_none()` return 1 if the corresponding entry does not exist.
- `pte_present()`, `pmd_present()` and `pgd_present()` return 1 if the corresponding page table entries have the `PRESENT` bit set.
- `pte_clear()`, `pmd_clear()` and `pgd_clear()` will clear the corresponding page table entry.
- `pmd_bad()` and `pgd_bad()` are used to check entries when passed as input parameters to functions that may change the value of the entries. Whether they return 1 varies between the few architectures that define these macros. However, for those that actually define it, making sure the page entry is marked as present and accessed are the two most important checks.

/arch/x86/include/asm/pgtable.h

Agenda

1. **Memory Addressing**
2. **Introduction to Linux Physical and Virtual Memory**
   a) **Linux Physical Memory**
   b) **Linux Virtual Memory**
3. **Allocators**
   a) **vmalloc(Noncontiguous memory area management)**
   b) **Physical Page Allocation**
   c) **sla/ub**
4. **Process address space**


UMA 架构
CPU0
CPU1
CPU2
CPU3
MC
4GB
NUMA 架构
CPU0
CPU1
CPU2
CPU3
CPU4
CPU5
CPU6
CPU7
MC
MC
4GB
4GB
内存域：
DMADMA32NORMAL HIGH
HIGH
node 0
node 1

Linux 中的物理内存

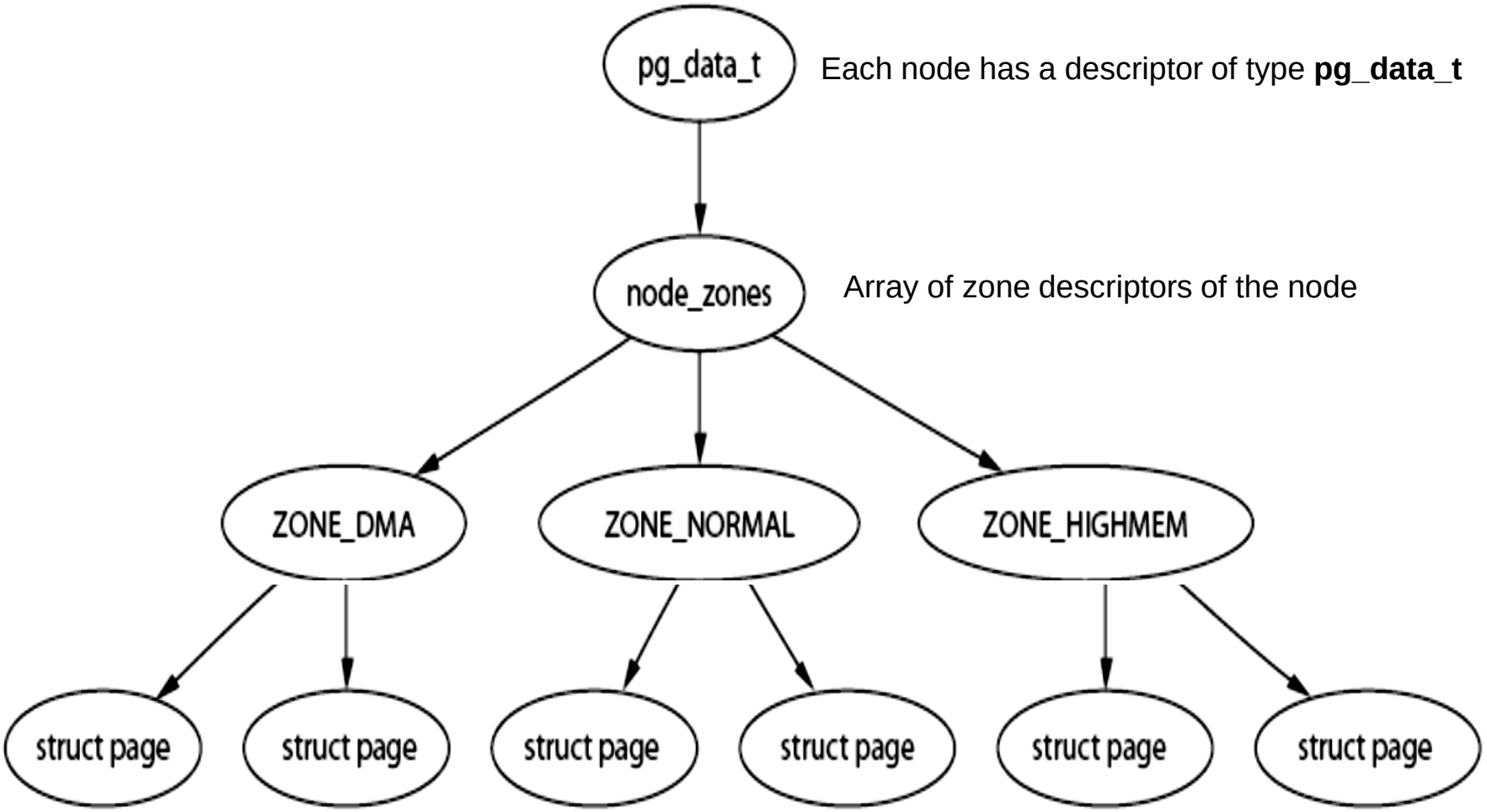
- **Linux** 的物理内存顶层数据结构用 **node** 表示，每个 **node** 代表和 **NUMA(Non Uniform Memory Access**非一致内存访问**)** 相关的一个内存 **bank**。
- 对于对称多处理器、**Multi-core** 处理器、一个**CPU** 的系统来说，只有一个 **node**。
- **X86**体系结构是**UMA**，本课不讨论**NUMA**

Nodes

- **Designed for NUMA (Non-Uniform Memory Access) machine.**
- **Each bank (The memory assigned to a CPU) is called a node and is represented by struct pglist_data.**

```
struct pglist_data *node_data[MAX_NUMNODES]
EXPORT_SYMBOL(node_data);
```

Relationship Between Nodes, Zones and Pages



<include/linux/mmzone.h>

```
typedef struct pglist_data {
        struct zone node_zones[MAX_NR_ZONES];
        struct node_zonelists[MAX_ZONELISTS];
        int nr_zones;
#ifdef CONFIG_FLAT_NODE_MEM_MAP   /* means !SPARSEMEM */
        struct page *node_mem_map;
#ifdef CONFIG_PAGE_EXTENSION
        struct page_ext *node_page_ext;
#endif
#endif
#ifndef CONFIG_NO_BOOTMEM
        struct bootmem_data *bdata;
#endif
#ifdef CONFIG_MEMORY_HOTPLUG
        spinlock_t node_size_lock;
#endif
        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
                                             range, including holes */
```

<include/linux/mmzone.h>

```
        int node_id;
        wait_queue_head_t kswapd_wait;
        wait_queue_head_t pfmemalloc_wait;
        struct task_struct *kswapd;   /* Protected by
                                mem_hotplug_begin/end() */
        int kswapd_max_order;
        enum zone_type classzone_idx;
#ifdef CONFIG_NUMA_BALANCING
        /* Lock serializing the migrate rate limiting window */
        spinlock_t numabalancing_migrate_lock;

        /* Rate limiting time interval */
        unsigned long numabalancing_migrate_next_window;

        /* Number of pages migrated during the rate limiting time interval */
        unsigned long numabalancing_migrate_nr_pages;
#endif
} pg_data_t;
```

The fields of the node descriptor

Type	Name	Description
`struct zone []`	**`node_zones`**	Array of zone descriptors of the node
`struct zonelist []`	`node_zonelists`	The order of zones that allocations are preferred from
`int`	`nr_zones`	Number of zones in the node
`struct page *`	**`node_mem_map`**	This is the first page of the `struct page` array that represents each physical frame in the node
`struct bootmem_data *`	**`bdata`**	Used by boot memory allocator during kernel initialization
`unsigned long`	`node_start_pfn`	The starting physical page frame number of the node
`unsigned long`	`node_present_pages`	Total number of physical pages in the node
`unsigned long`	`node_spanned_pages`	Total size of physical page range, including holes
`int`	`node_id`	Node ID (NID) of the node

Zones

- 每个 **node** 划分为多个区域**(zone)**，对具有不同性质的页进行分组
- 一般有**3**个区，每个区域的物理内存可以是不连续的。
  - ✓ **ZONE_DMA----**一些硬件只能对特定地址范围执行**DMA**操作。
  - ✓ **ZONE_NORMAL ----** 这个区包含的都是能够正常映射的页。
  - ✓ **ZONE_HIGHMEM ----** 一些体系架构不能将所有物理页映射到内核空间，则通过高端内存解决此问题。其中的页并不能永久地映射到内核地址空间，是动态映射的页。
- **Linux** 内核在启动的时候根据 **BIOS** 提供的信息来设置**node**和 **zone**相关数据信息。

Zones

X86-32		X86-64	
ZONE_DMA	0-16MB	ZONE_DMA	0-16MB
ZONE_NORMAL	16MB-896MB	ZONE_DMA32	16Mb-4GB
ZONE_HIGHMEM	896MB-	ZONE_NORMAL	4GB-END

- **ZONE_DMA 和 ZONE_NORMAL 是可以直接访问的。**
- **ZONE_HIGHMEM 包含内核不能直接访问的页面，在 64-bit 的体系结构中，ZONE_HIGHMEM 为空。**

Struct zone

/include/linux/mmzone.h

Three zones in Linux, described by struct zone

```
struct zone {
  …
   unsigned long watermark[NR_WMARK];
  …
   long lowmem_reserve[MAX_NR_ZONES];
  …
   unsigned int inactive_ratio;

   struct pglist_data   *zone_pgdat;
   struct per_cpu_pageset __percpu *pageset;
  …
    unsigned long       dirty_balance_reserve;
  …
     unsigned long      zone_start_pfn;
 …
   unsigned long        managed_pages;
   unsigned long        spanned_pages;
   unsigned long        present_pages;

   const char         *name;
 …
    int             nr_migrate_reserve_block;
```

```
enum zone_watermarks {
        WMARK_MIN,
        WMARK_LOW,
        WMARK_HIGH,
        NR_WMARK
};
```

Struct zone

```
    ...
    wait_queue_head_t   *wait_table;
    unsigned long       wait_table_hash_nr_entries;
    unsigned long       wait_table_bits;

    ZONE_PADDING(_pad1_)
    ...
    struct free_area    free_area[MAX_ORDER];

   ...
    unsigned long       flags;

   ...
    spinlock_t          lock;

    ZONE_PADDING(_pad2_)
  ...
     spinlock_t         lru_lock;
    struct lruvec       lruvec;
  ...
    atomic_long_t       inactive_age;
  ...
   unsigned long percpu_drift_mark;
  ...
    ZONE_PADDING(_pad3_)
    atomic_long_t       vm_stat[NR_VM_ZONE_STAT_ITEMS];
}
```

The fields of the zone descriptor

Type	Name	Description
struct free_area	free_area	Free area bitmaps used by the buddy allocator
wait_queue_head_t*	wait_table	a hash table of wait queue of process waiting on a page to be freed
unsigned long	wait_table_hash_nr_entries	the size of the hash table array
unsigned long	wait_table_bits	the number of bits in a page address from left to right being used as an index within the wait_table
struct per_cpu_pageset	pageset	per CPU pageset for order-0-page allocation(to avoid interrupt-safe spinlock on SMP system)
struct pglist_data*	zone_pgdat	points to the descriptor of the parent node
unsigned long	zone_start pfn	the starting physical page frame number of the zone
const char*	name	The string name of the zone : ” DMA ”,” Normal ” or “ HighMem ”
unsigned long	managed pages	the total pages spanned by the zone, including holes
unsigned long	spanned pages	physical pages existing within the zone
unsigned long	present pages	present pages managed by the buddy system

PAGE

- Linux 内存管理是页式管理,物理页是内存分配的基本单位。
- 每个node里的pages都在一个链表上；
- 根据页的使用情况，每个页可能在buddy分配器，以及LRU链表、address_space以及slab相关的链表上

Pages

- 每个物理页都有一个**页结构**与之对应。
- Kernel在启动的时候，初始化所有的页结构（page structure）。
- To keep track of all physical pages, all physical pages are described by an array of struct page called **mem_map.**

/linux/mm/memory.c

```
struct page *mem_map;
EXPORT_SYMBOL(mem_map);
```

```
typedef struct pglist_data {
……
#ifdef CONFIG_FLAT_NODE_MEM_MAP
    struct page *node_mem_map;  /* means !SPARSEMEM */
```

struct page

/include/linux/mm_types.h

```
struct page {
    unsigned long flags;
    union {
        struct {      /* Page cache and anonymous pages */
            struct list_head lru;
            struct address_space *mapping;
            pgoff_t index;
            unsigned long private;          };
        ……
        struct {  /* slab, slob and slub */
            union {
                struct list_head slab_list;
                struct {/* Partial pages */
                    struct page *next;
#ifdef CONFIG_64BIT
                    int pages;  /* Nr of pages left */
                    int pobjects;    /* Approximate count */
            …… }; …… };  ……}; ……};  ……} _struct_page_alignment;
```

The fields of the page descriptor

Type	Name	Description
page_flag_t	flags	The status of the page and mapping of the page to a zone
atomic_t	_count	The reference count to the page. If it drops to zero, it may be freed
unsigned long	private	Mapping private opaque data: usually used for buffer_heads if PagePrivate set
struct address_space *	mapping	Points to the address space of a inode when files or devices are memory mapped.
pgoff_t	index	Our offset within mapping
struct list_head	lru	Linked to LRU lists of pages if the page is in page cache Linked to free_area lists if the page is free and is managed by buddy allocator

Page->flags {linuxsrcdir}/include/linux/page_flags.h

```
enum pageflags {
  PG_locked,          /* Page is locked. Don't touch. */
  PG_error,
  PG_referenced,
  PG_uptodate,
  PG_dirty,
  PG_lru,
  PG_active,
  PG_slab,
  PG_owner_priv_1,  /* Owner use. If pagecache, fs may use*/
  PG_arch_1,
  PG_reserved,
  PG_private,         /* If pagecache, has fs-private data */
  PG_private_2,       /* If pagecache, has fs aux data */
  PG_writeback,       /* Page is under writeback */
  ....
}
```

设置和清除这些 flags 的一些宏

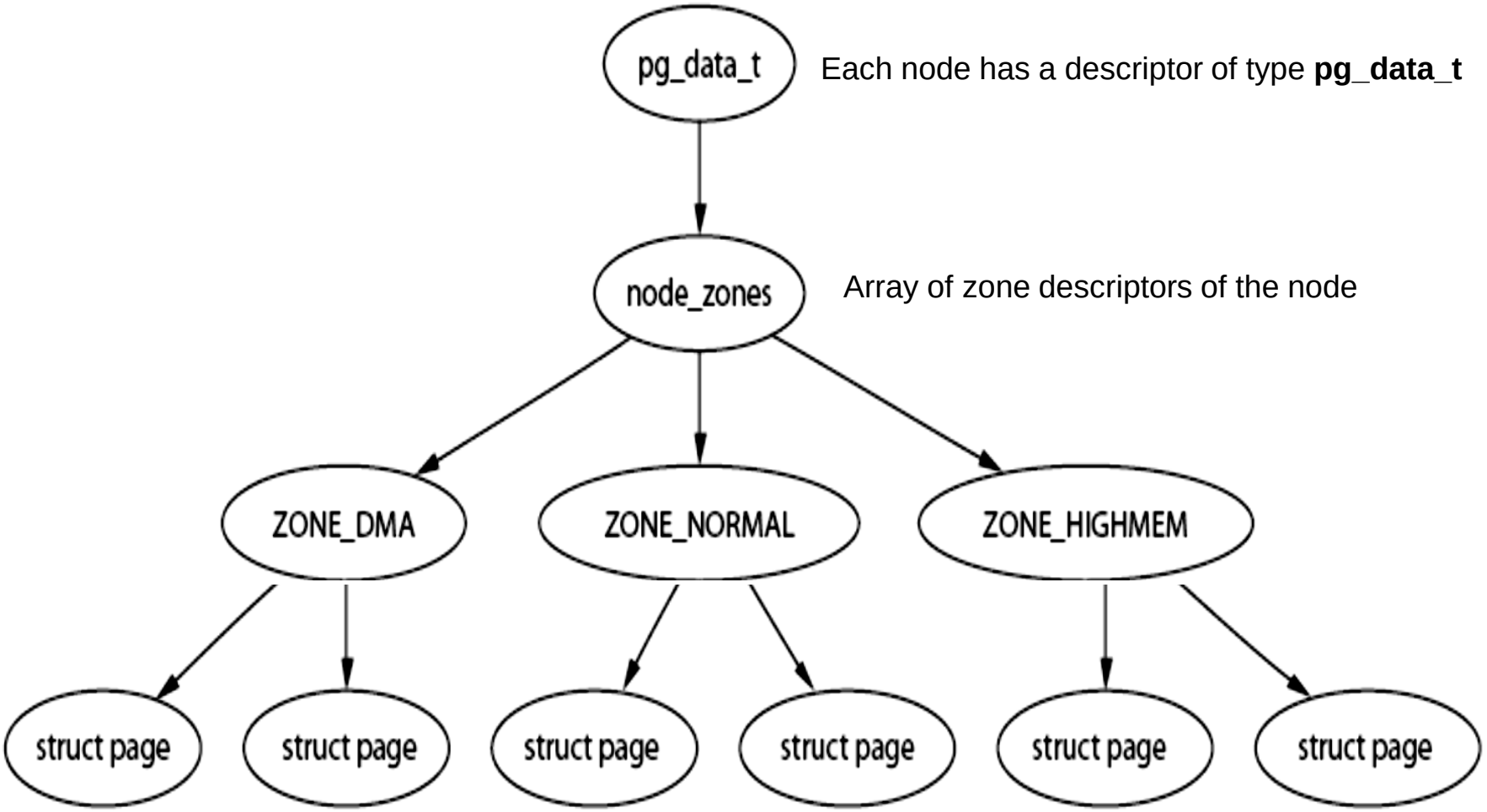
{linuxsrcdir}/include/linux/page_flags.h

```
/*设置flag*/
#define SETPAGEFLAG(uname, lname, policy)                       \
static __always_inline void SetPage##uname(struct page *page)          \
    { set_bit(PG_##lname, &policy(page, 1)->flags); }

/*清除flag*/
#define CLEARPAGEFLAG(uname, lname, policy)                     \
static __always_inline void ClearPage##uname(struct page *page)        \
    { clear_bit(PG_##lname, &policy(page, 1)->flags); }
```

{linuxsrcdir}/ arch/x86/include/asm/bitops.h

```
static inline void set_bit(int nr, void *addr)
{
    asm("btsl %1,%0" : "+m" (*(u32 *)addr) : "Ir" (nr));
}
```

Relationship Between Nodes, Zones and Pages



物理内存map

/linux/System.map
01000000 A phys_startup_32

- 有些物理页面是被内核以及其他数据结构所占有，要标记为保留，不属于动态分配的范围。
  - **Linux 内核在 x86-32 体系结构下被加载在 RAM 的 0x001000000地址处。**
- 为什么不把内核加载到 **RAM 的 0 地址处呢？**
  - **Page frame 0 被 BIOS 存放了由 Power-On Self-Test(POST) 产生的系统硬件配置信息。**
  - **从 0x000a0000 到 0x000fffff 的地址保留给 BIOS 例程，用来映射ISA设备(比如显卡)的内部内存，操作系统不能分配相关的物理页。**
  - 还有一些页也为某些特殊的机型保留。

Cont'

在启动阶段，内核就得到了所有的物理上能用的内存的一个 layout，包括大小、范围等，可在启动汇编代码中看到得到这些信息的方法。

Table 2-9. Example of BIOS-provided physical addresses map

Start	End	Type
0x00000000	0x0009ffff	Usable
0x000f0000	0x000fffff	Reserved
0x00100000	0x07feffff	Usable
0x07ff0000	0x07ff2fff	ACPI data
0x07ff3000	0x07ffffff	ACPI NVS
0xffff0000	0xffffffff	Reserved

BIOS中断号15H，AX=E820H

Agenda

1. **Memory Addressing**
2. **Introduction to Linux Physical and Virtual Memory**
   a) **Linux Physical Memory**
   b) **Linux Virtual Memory**
3. **Allocators**
   a) **vmalloc(Noncontiguous memory area management)**
   b) **Physical Page Allocation**
   c) **sla/ub**
4. **Process address space**

虚拟内存(Virtual Memory)

- 虚拟内存可以让每个进程都有自己的虚拟地址空间，即自己的线性内存地址空间
- 硬件设施 **MMU (memory management unit)** 负责把虚拟地址（线性地址）转化为物理地址。
- **TLB：Translation lookaside buffer**，缓存页表数据
- 在 **Linux** 中, 内核和用户地址空间在同一个虚拟地址空间中。
- 在**x86-32**平台上，虚拟内存地址空间分为两部分： **0-3G** 是用户空间；**3G-4G** 是内核地址空间。
- 用户程序可以使用系统调用接口：**mmap**以及**brk** 来分配虚拟内存空间。

线性地址空间

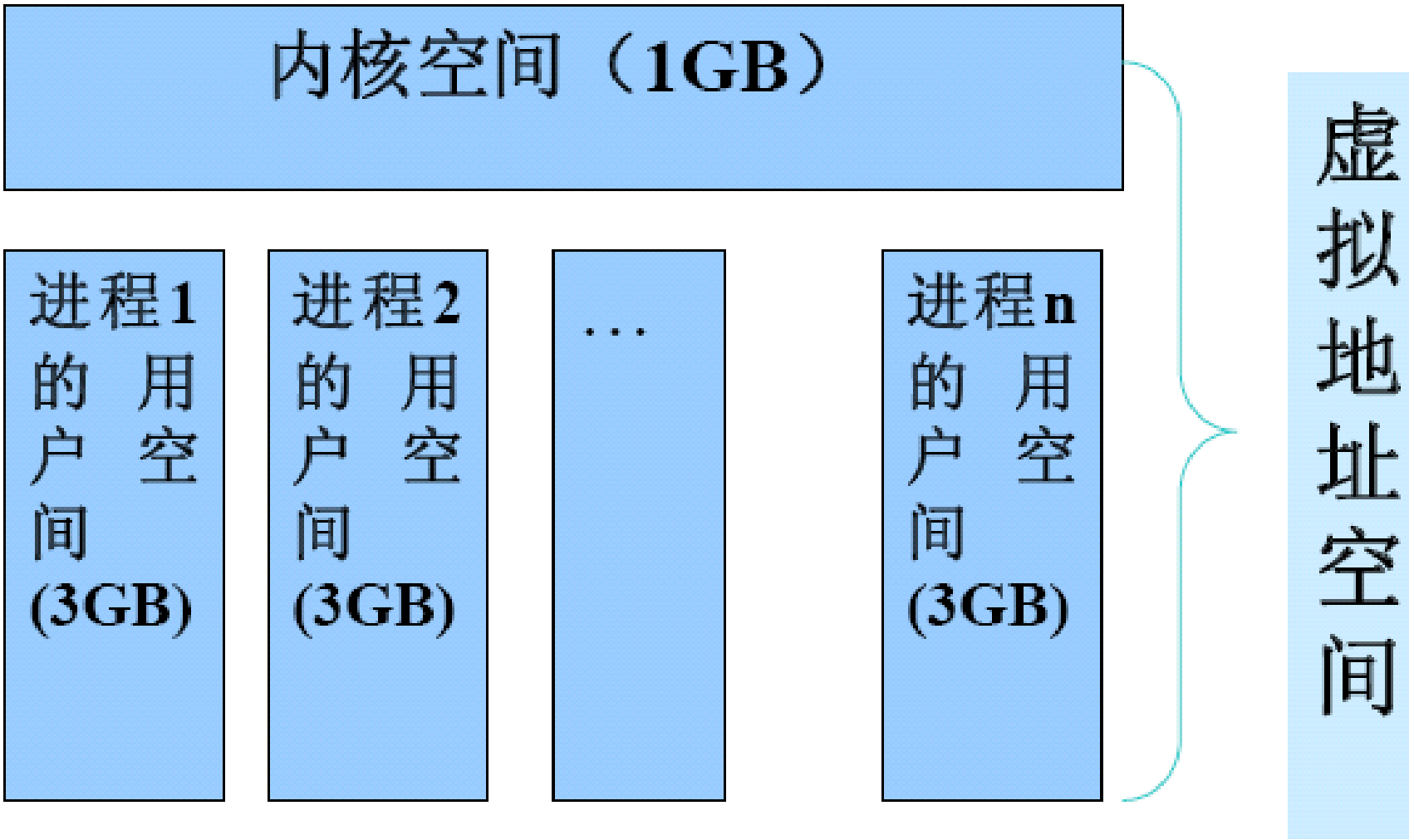
X86-32

- 不论用户态下的进程还是内核态下的进程都可以访问 **0x00000000** 到 **0xbfffffff** 线性地址范围。
- 从**0xc0000000** 到 **0xffffffff** 的线性地址只能由在内核态下运行的进程访问。
- 当一个进程在用户态下运行，它的执行地址空间小于 **0xc0000000**；但当其切换至内核地址空间执行时，它的地址空间就等于或大于 **0xc0000000**。
- 宏**PAGE_OFFSET = 0xc0000000**，是进程在线性地址空间中的偏移量，也是内核地址空间的开始之处。

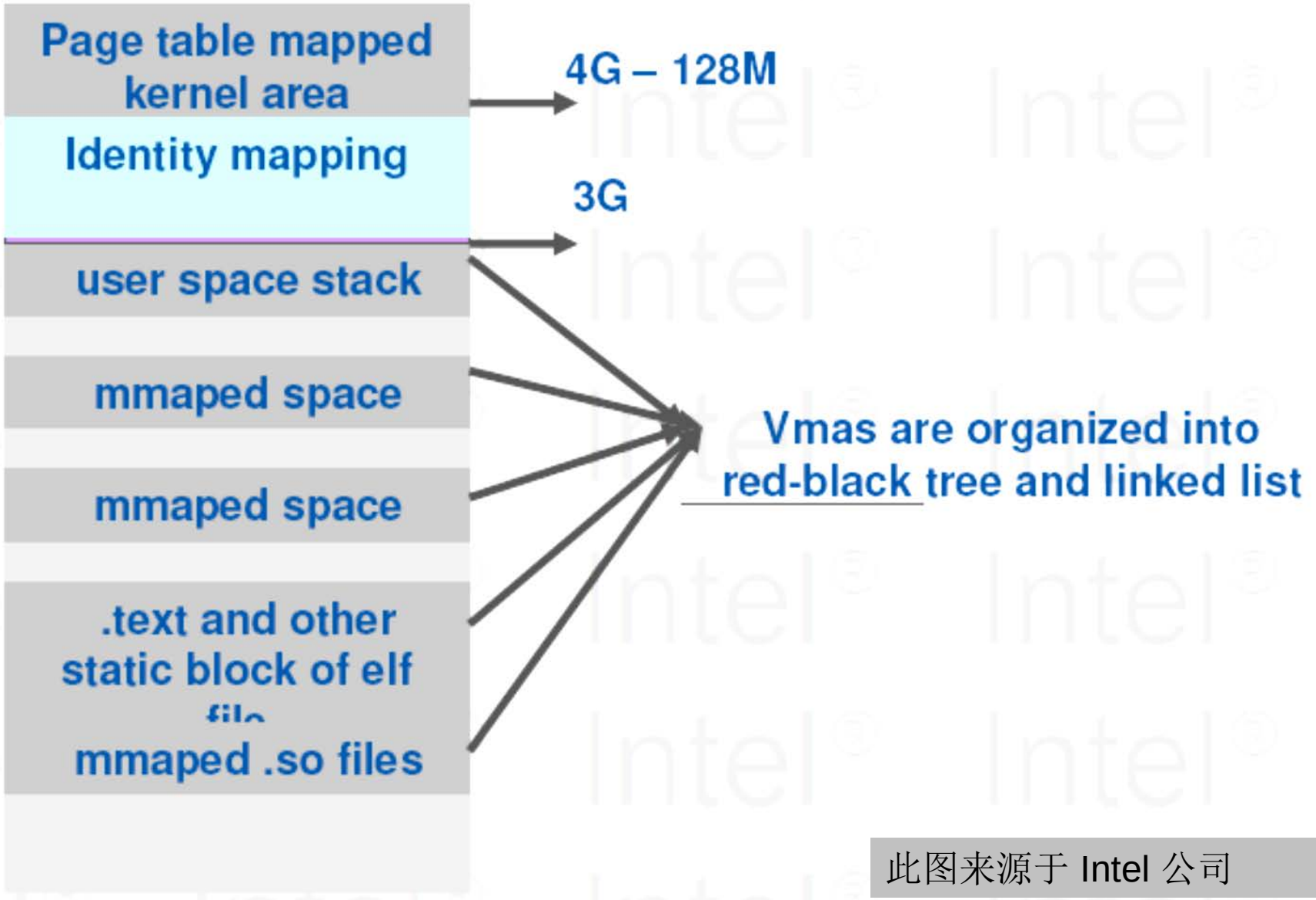
/linux/arch/x86/include/asm/page_32_types.h
#define __START_KERNEL_map __PAGE_OFFSET

内核页表

- 供内核自己使用，称为“主内核页全局目录”（**master kernel Page Global Directory**）。
- 把主内核页全局目录的最高目录项部分作为参考模型，提供给系统中每个普通进程对应的页全局目录项。


内核空间（1GB）
进程1的用户空间(3GB)
进程2的用户空间(3GB)
…
进程n的用户空间(3GB)
虚拟地址空间

用户空间的虚拟内存

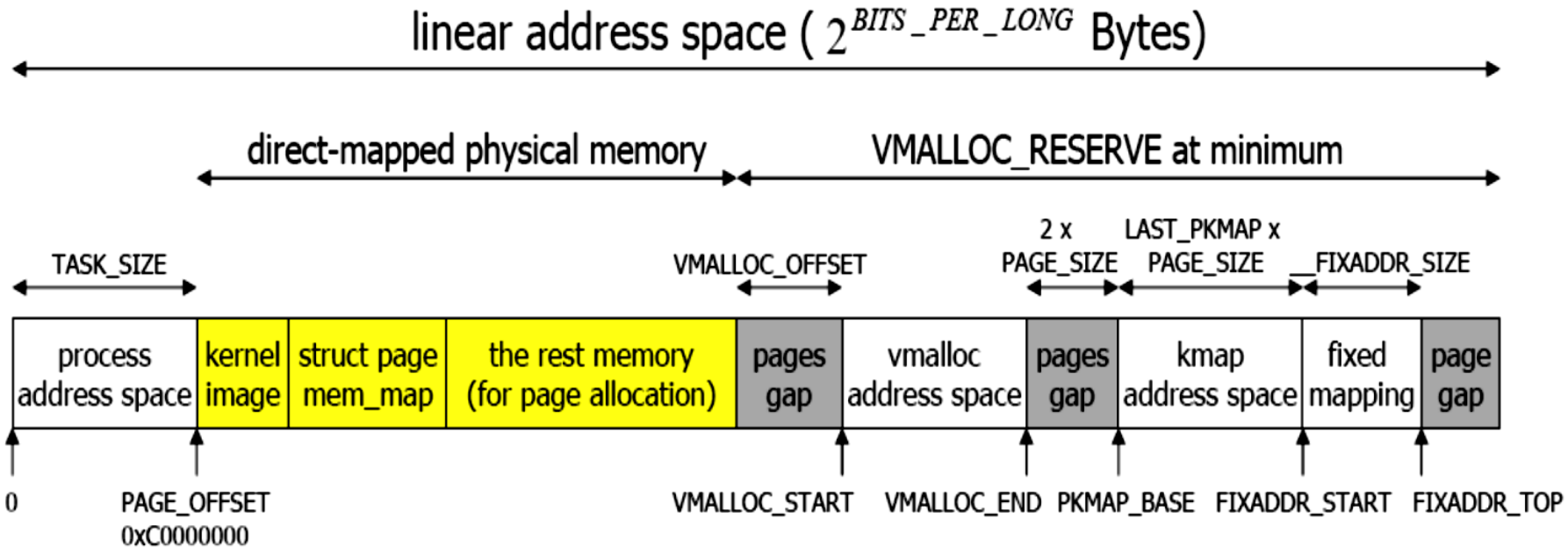


此图来源于 Intel 公司

Cont'

- 内核通过“请求调页（**demand paging**）”可以提供比物理内存空间更多的虚拟内存地址空间。
- 在 **task_struct** 中有一个 **mm_struct** 来指向任务的虚拟内存空间，在同一进程中的不同的线程，使用同一个 **mm_struct**。
- 内核线程的 **mm_struct** 是 **NULL**。
- 在 **mm_struct** 中有一 **vm_struct vma** 链表的红-黑树**(red-black tree)**，这些 **vma** 也被组织成 **sorted list**。

内核虚拟地址空间



- vmalloc address space
  - Noncontiguous physical memory allocation
- kmap address space
  - Allocation of memory from ZONE_HIGHMEM
- Fixed mapping
  - Compile-time virtual memory allocation

此图来源：http://www.cs.ccu.edu.tw/~lhr89/linux-kernel/Physical%20Memory%20Management%20in%20Linux.pdf

Cont'

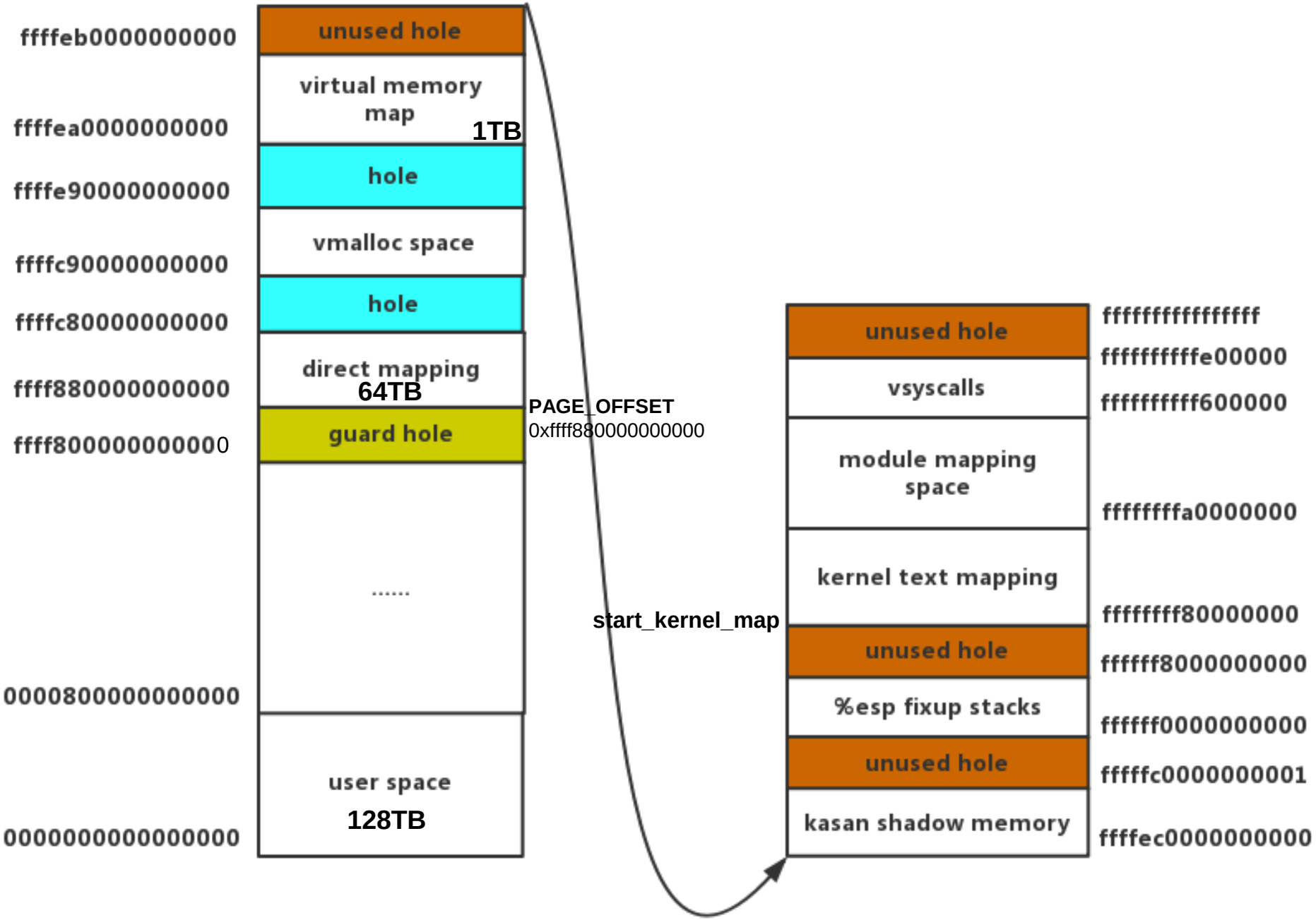
x86_64

- 不论用户态下的进程还是内核态下的进程都可以访问**0x0000000000000000**到**0x00007fffffffffff** 线性地址范围。
- 从**0xffff800000000000**到 **0xffffffffffffffff** 的线性地址只能由在内核态下运行的进程访问。
- 宏**PAGE_OFFSET =0xffff880000000000**,内核地址空间映射开始之处
- 宏**__START_KERNEL_map=0xffffffff80000000**是内核代码映射开始之处

/linux/arch/x86/include/asm/page_64_types.h

```
#define __PAGE_OFFSET           _AC(0xffff880000000000, UL)
#define __START_KERNEL_map   _AC(0xffffffff80000000, UL)
```

内核虚拟地址空间(x86_64)



directly mapped kernel virtual address Translation

- memory in ZONE_DMA and ZONE_NORMAL is direct-mapped and all page frames are described by mem_map array.
- Kernel virtual address -> **physical address**
- Kernel virtual address -> **struct page**
  - Use physical address as an index into the mem_map array

linux/arch/x86/include/asm/io.h

```
static inline phys_addr_t virt_to_phys(volatile void * address)
{
    return __pa(address);
}
```

linux/arch/x86/include/asm/page.h

```
#define __pa(x) __phys_addr((unsigned long)(x))
#define virt_to_page(kaddr) pfn_to_page(__pa(kaddr) >> PAGE_SHIFT)
```

linux/arch/x86/include/asm/page_32.h

```
#define phys_addr(x)
__phys_addr_nodebug(x)
#define __phys_addr_nodebug(x) ((x) -
PAGE_OFFSET)
```

The pfn_to_page(pfn) macro yields the address of the page descriptor associated with the page frame having number pfn.

Agenda

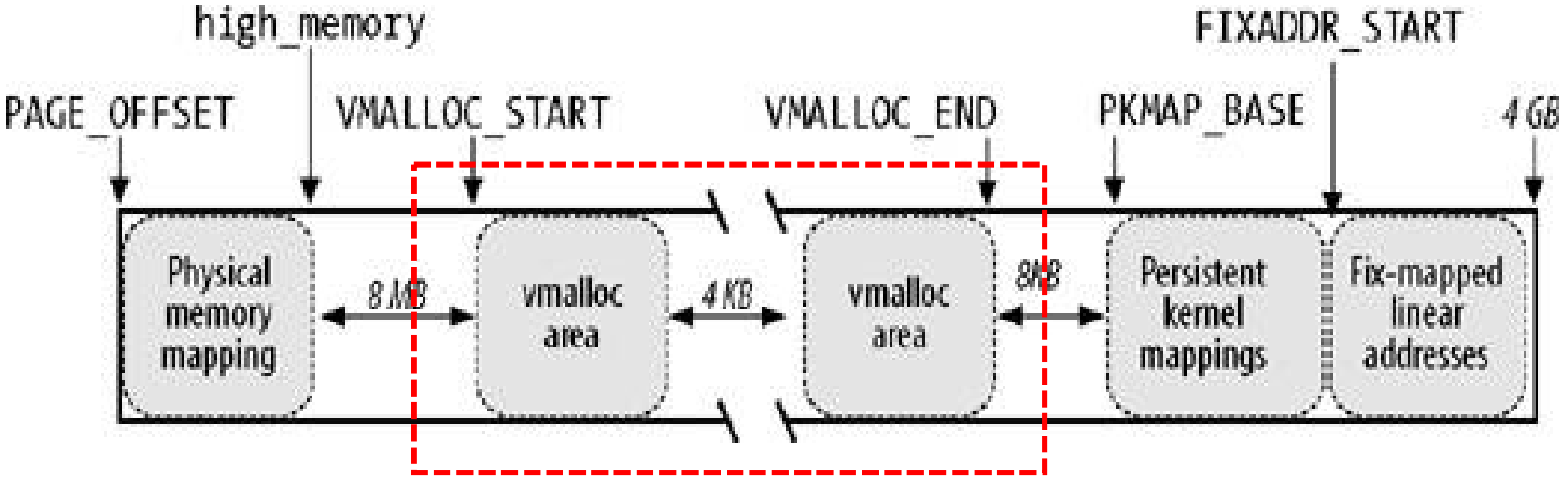
1. **Memory Addressing**
2. **Introduction to Linux Physical and Virtual Memory**
3. **Allocators**
   a) **vmalloc(Noncontiguous memory area management)**
   b) **Physical Page Allocation**
   c) **sla/ub**
4. **Process address space**

不同层次的allocator

	Description	Used at	functions
Boot Memory Allocator	1. A first-fit allocator, to allocate and **free** memory during kernel boots 2. Can handle allocations of sizes smaller than a page	System boot time	`alloc_bootmem()` `free_bootmem()`
Physical Page Allocator (buddy system)	1. **Page-size** physical frame management 2. Good at dealing with **external fragmentation**	After `mem_init()`, at which boot memory allocator retires	`alloc_pages()` `__get_free_pages()`
Slab Allocator	1. Deal with **Internal fragmentation** (**for allocations < page-size**) 2. Caching of commonly used objects 3. Better use of the hardware cache	After `mem_init()`, at which boot memory allocator retires	`kmalloc()` `kfree()`
Virtual Memory Allocator	1. Built on top of page allocator and **map noncontiguous physical pages to logically contiguous vmalloc space** 2. Required altering the kernel page table 3. Size of all allocations <= vmalloc address space	1. Large allocation size 2. contiguous physical memory is not available	`vmalloc()` `vfree()`

此图来源：http://www.cs.ccu.edu.tw/~lhr89/linux-kernel/Physical%20Memory%20Management%20in%20Linux.pdf

Linear Address of Noncontiguous Memory Areas



不连续**memory area** 应用：
1.不经常使用的内存映射（经常使用的内存的例子是：经常反复分配、回收的内存）；
2.为 **swap** 区域分配数据结构；
3.为某些**I/O** 设备驱动程序分配缓冲区；
4.利用 **high memory page frames**；
5.可以避免一些外部碎片**(external fragmentation)**。

- 使用不连续内存区域需要和内核页表**(kernel page table)**打交道，对内核页表有创建、释放等操作。
- 图上的**8M,4k,8k**表示“**safety interval**”. 目的是用来”**capture**” **out-of-bounds memory access.**

vm_struct

```
struct vm_struct {
    struct vm_struct    *next;      /*指向下一个vm区域*/
    void                *addr;      /*指向第一个内存单元(线性地址)*/
    unsigned long       size;       /*该块内存区的大小*/
    unsigned long       flags;      /*内存类型的标识字段*/
    struct page         **pages;    /*指向页描述符指针数组*/
    unsigned int        nr_pages;   /*内存区大小对应的页框数*/
    unsigned long       phys_addr;  /*用来映射硬件设备的IO共
                                      享内存，其他情况下为0*/
    void                *caller;    /*调用vmalloc类的函数的返回地址*/
};
```

/include/linux/vmalloc.h

Descriptors of Noncontiguous Memory Areas (vm_struct)

Type	Name	Description
void*	addr	Linear address of the first memory cell of area
unsigned long	size	Size of the area plus 4,096(inter-area safety interval)
unsigned long	flags	Type of memory mapped by the noncontiguous memory area
struct page**	pages	Pointer to array of nr_pages pointer to page descriptors
unsigned int	nr_pages	Number of pages filled by area
phys_addr_t	phys_addr	Set to 0 unless the area has been created to map the I/O shared memory of a hardware device
struct vm_struct*	next	Pointer to next vm_struct structure
void	caller	Pointer to the function which calls vm_struct

Flags: 内存区类型；
VM_ALLOC: 由**vmalloc**产生；
VM_MAP：由**vmap**映射现有的**pages**集合；
VM_IOREMAP：由**ioremap**映射硬件设备内存。

```
struct vmap_area {
    unsigned long va_start; /*vmalloc区的起始地址*/
    unsigned long va_end; /*vmalloc区的结束地址*/
    unsigned long flags; /*类型标识*/
    struct rb_node rb_node;         /* address sorted rbtree */
    struct list_head list;          /* address sorted list */
    struct list_head purge_list;    /* "lazy purge" list */
    struct vm_struct *vm;
    struct rcu_head rcu_head;
};
```

vmalloc.c

```
void *vmalloc(unsigned long size)
{
        return __vmalloc_node_flags(size, NUMA_NO_NODE,
GFP_KERNEL | __GFP_HIGHMEM);
}
static inline void *__vmalloc_node_flags(unsigned long size,
                                int node, gfp_t flags)
{
        return __vmalloc_node(size, 1, flags, PAGE_KERNEL, node,
__builtin_return_address(0));
}
static void *__vmalloc_node(unsigned long size, unsigned long align,
                            gfp_t gfp_mask, pgprot_t prot,  int node, void *caller)
{
        return __vmalloc_node_range(size, align, VMALLOC_START,
MALLOC_END, gfp_mask, prot, node, caller);
}
```

```
void *__vmalloc_node_range(unsigned long size, unsigned long align, unsigned long
        start, unsigned long end, gfp_t gfp_mask, pgprot_t prot, int node, void *caller)
{       struct vm_struct *area;
        void *addr;
        unsigned long real_size = size;
        size = PAGE_ALIGN(size);
        if (!size || (size >> PAGE_SHIFT) > totalram_pages)
                goto fail;
        area = __get_vm_area_node(size, align, VM_ALLOC | VM_UNLIST,
                                start, end, node, gfp_mask, caller);
        if (!area)
                goto fail;
        addr = __vmalloc_area_node(area, gfp_mask, prot, node);
        if (!addr)
                return NULL;
          /*
         * In this function, newly allocated vm_struct has VM_UNINITIALIZED
         * flag. It means that vm_struct is not fully initialized.
         * Now, it is fully initialized, so remove this flag here.
         */
        clear_vm_uninitialized_flag(area);
        kmemleak_alloc(addr, real_size, 2, gfp_mask);
        return addr;
fail:
        warn_alloc_failed(gfp_mask, 0, "vmalloc: allocation failure: %lu bytes\n", real_size);
        return NULL;
}
```

分配area数据结构
分配vm区域

分配页框
并作映射

vmalloc.c

分配vm_struct及vmap_area并初始化，分配虚拟空间 vmalloc.c

```
static struct vm_struct *__get_vm_area_node(unsigned long size,
        unsigned long align, unsigned long flags, unsigned long start,
        unsigned long end, int node, gfp_t gfp_mask, void *caller)
{
    struct vmap_area *va;
    struct vm_struct *area;

    BUG_ON(in_interrupt()); //中断不能调用，因为kzalloc可能休眠
    if (flags & VM_IOREMAP) {
        int bit = fls(size);

        if (bit > IOREMAP_MAX_ORDER)
            bit = IOREMAP_MAX_ORDER;
        else if (bit < PAGE_SHIFT)
            bit = PAGE_SHIFT;

        align = 1ul << bit;
    }

    size = PAGE_ALIGN(size); //大小页对齐
    if (unlikely(!size))
        return NULL;
```

```
        area = kzalloc_node(sizeof(*area), gfp_mask &
        GFP_RECLAIM_MASK, node); // 分配vm_struct对象并清零

    if (unlikely(!area))
        return NULL;

    /*
     * We always allocate a guard page.
     */
    size += PAGE_SIZE; // with gap

    //分配虚拟区域，并插入链表
    va = alloc_vmap_area(size, align, start, end, node, gfp_mask);
    if (IS_ERR(va)) {
        kfree(area);
        return NULL;
    }

    setup_vmalloc_vm(area, va, flags, caller); //初始化area

    return area;
}
```

//分配页框并作映射 vmalloc.c

```
static void *__vmalloc_area_node(struct vm_struct *area, gfp_t gfp_mask,
                                 pgprot_t prot, int node, void *caller)
{
    const int order = 0;
    struct page **pages;
    unsigned int nr_pages, array_size, i;
    gfp_t nested_gfp = (gfp_mask & GFP_RECLAIM_MASK) | __GFP_ZERO;

        //内存区大小对应的页框数
    nr_pages = (area->size - PAGE_SIZE) >> PAGE_SHIFT;
    array_size = (nr_pages * sizeof(struct page *)); //页框指针区域多大

    area->nr_pages = nr_pages;
    /* Please note that the recursion is strictly bounded. */
    //分配页框指针区域
    if (array_size > PAGE_SIZE) {
        pages = __vmalloc_node(array_size, 1,
nested_gfp|__GFP_HIGHMEM, PAGE_KERNEL, node, caller);
        area->flags |= VM_VPAGES;
    } else {
        pages = kmalloc_node(array_size, nested_gfp, node);
    }
```

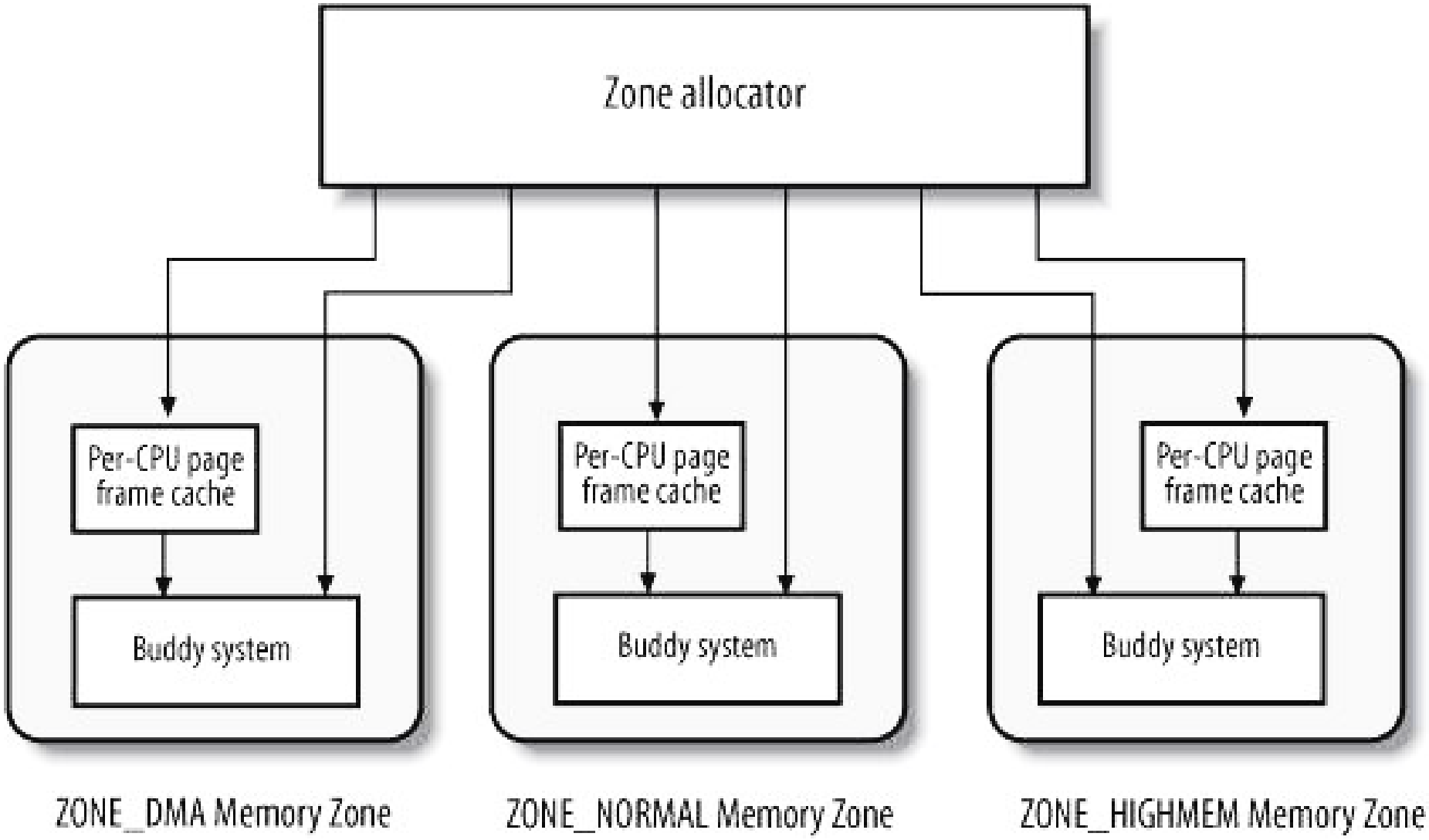
```
area->pages = pages;
if (!area->pages) {
        remove_vm_area(area->addr);
        kfree(area);
        return NULL;
}
        //分配物理页框
        for (i = 0; i < area->nr_pages; i++) {
        struct page *page;
        gfp_t tmp_mask = gfp_mask | __GFP_NOWARN;
                        /* 没有指定在哪个NUMA节点中分配页面 */
        if (node < NUMA_NO_NODE)
                page = alloc_page(tmp_mask); ///* 随便分配一个页面 */
        else
                                /* 在指定NUMA节点中分配页面 */
                page = alloc_pages_node(node, tmp_mask, order);

        if (unlikely(!page)) {
                /* Successfully allocated i pages, free them in __vunmap() */
```

```
        area->nr_pages = i;
                goto fail;
        }
        area->pages[i] = page;
    }

    if (map_vm_area(area, prot, &pages)) //建立页表映射
        goto fail;
    return area->addr;

fail:
    warn_alloc_failed(gfp_mask, order,
                      "vmalloc: allocation failure, allocated %ld
of %ld bytes\n",
                       (area->nr_pages*PAGE_SIZE), area-
>size);
    vfree(area->addr);
    return NULL;
}
```

Agenda

1. **Memory Addressing**
2. **Introduction to Linux Physical and Virtual Memory**
3. **Allocators**
   a) **vmalloc(Noncontiguous memory area management)**
   b) **Physical Page Allocation**
   c) **sla/ub**
4. **Process address space**

Components of the zoned page frame allocator



Zone Allocator

管理区分配器是内核页框分配器的前端，分配一个包含足够多空闲页框的内存管理区，使它能满足内存请求。管理区分配器必须满足几个目标：

- 它应当保护保留的页框池（**reserved pool**）。
- 当内存不足且允许阻塞当前进程时， 它应当触发页框回收算法。
- 一旦某些页框被释放，管理区分配器将再次尝试分配。
- 如果可能，它应当尽可能保存小而珍贵的 **ZONE_DMA** 内存管理区。

Reserved pool

- 当请求内存时，一些内核控制路径不能被阻塞（例如处理中断）。此时，内核控制路径应当产生原子内存分配请求（使用**GFP_ATOMIC**标志）。
- 原子请求从不被阻塞：如果没有足够的空闲页，则仅仅是分配失败而已。
- 内核会设法尽量减少这种不幸事件发生的可能性——**Reserved Pool**，只有在内存不足时才使用。
- 存放在变量**min_free_kbytes**

```
mm/page_alloc.c
int __meminit init_per_zone_wmark_min(void)
{……
        lowmem_kbytes = nr_free_buffer_pages() * (PAGE_SIZE >> 10);
        new_min_free_kbytes = int_sqrt(lowmem_kbytes * 16);
    ……  }
```

```
unsigned long nr_free_buffer_pages(void)
{
	return nr_free_zone_pages(gfp_zone(GFP_USER));
}

static unsigned long nr_free_zone_pages(int offset)	/*count number of pages
{                                                       beyond high watermark*/
	struct zoneref *z;
	struct zone *zone;

	/* Just pick one node, since fallback list is circular */
	unsigned long sum = 0;

	struct zonelist *zonelist = node_zonelist(numa_node_id(), GFP_KERNEL);

	for_each_zone_zonelist(zone, z, zonelist, offset) {
		unsigned long size = zone_managed_pages(zone);
		unsigned long high = high_wmark_pages(zone);
		if (size > high)
			sum += size - high;
	}

	return sum;
}
```

Physical Page Allocation

Binary Buddy Allocator(二进制伙伴分配器)

- 所有的页框分**11**组，每组分别包括**1，2，4，8，16, 32, 64, 128, 256, 512, 1024** 个连续的页面。
- 伙伴物理内存管理的基本思想：
  - 分配内存的时候，先从某一组中找能满足要求的块，如果有，分配空闲块；如果没有，从包含更多页面的组中找空闲块，这时，对包含更多页面的空闲块分割，并把分割后的剩余的块，根据各组支持的块的页的数目重新插入到相应的组中。
  - 释放内存块的时候，根据与其相邻的内存块的使用情况和大小进行合并，重新插入到相应的组中。

对于 Linux 内核而言

- **Allocate from buddy system**
  - **if it is order 0 page, try to get page from the per-cpu pcp list.**
  - **if the pcp list empty, kernel will try to grab some pages from per zone free_area list in bulk and put them to per-cpu pcp list.**
  - **if it is order > 0 page, kernel will direct grab pages from per zone free_area.**
  - **if kernel is failed to allocate pages, it will try to do vm balancing.**
- **Free to buddy system**
  - **if it is an order 0 page, kernel will put page to per-cpu pcp list**
  - **if it is a order > 0 page, kernel will direct put the page back to zone free_area list**
  - **kernel will also try to merge free_area list to higher order list.**

each memory zone defines a **per-CPU page frame cache**. Each per-CPU cache includes some pre-allocated page frames to be used for single memory requests issued by the local CPU.

Manage Free Blocks

- 每个 **struct zone** 都有一个域是：

  struct free_area free_area[MAX_ORDER];

  #define MAX_ORDER 11

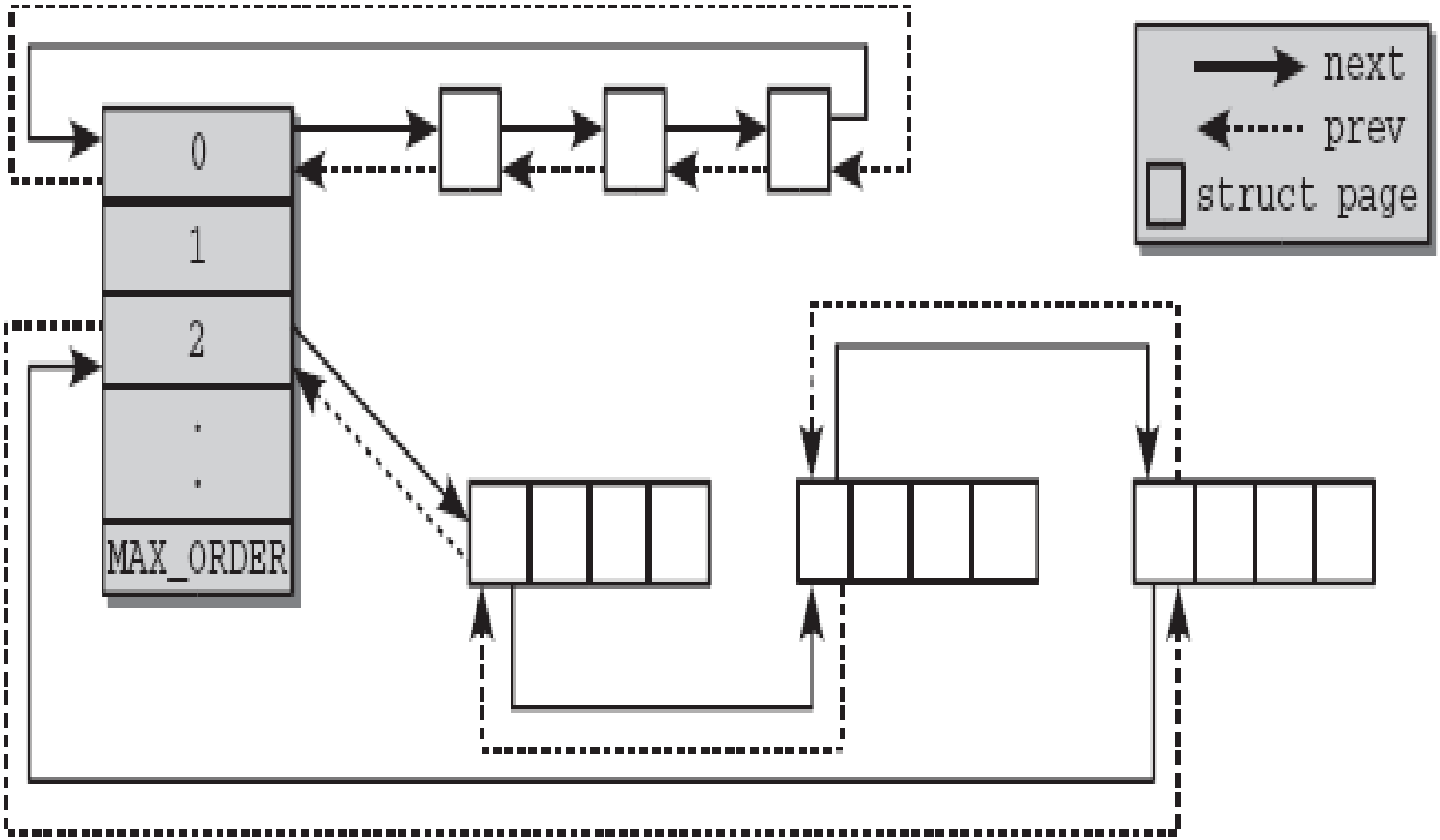
  Order: 2的幂次，表示内存块的大小（以页为单位）。如 2^0 表示只有一页大小的内存块

include/linux/mmzone.h

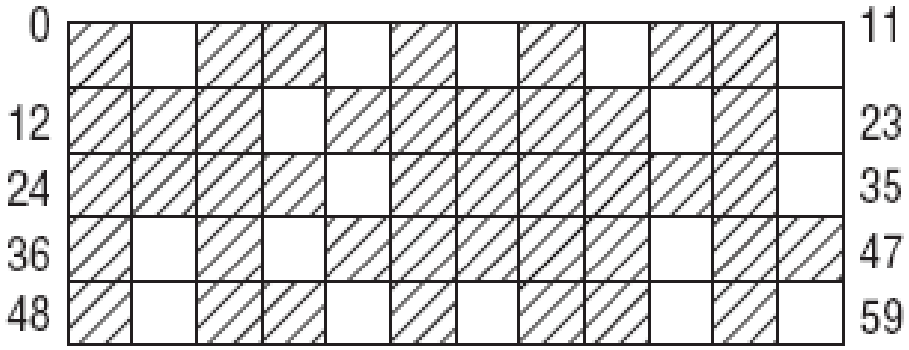
```
struct free_area {
        struct list_head        free_list[MIGRATE_TYPES];
        unsigned long           nr_free; //空闲页块的个数，
                                         //不是空闲页的个数
};
```

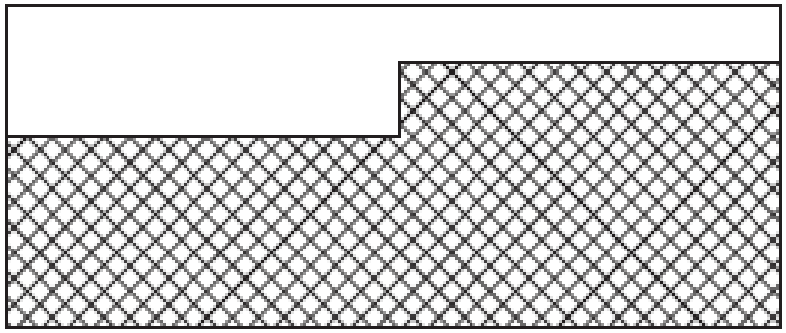
/proc/buddyinfo：各**zone**中不同大小空闲页块的个数。

Zone中的free_area以及Order



碎片问题



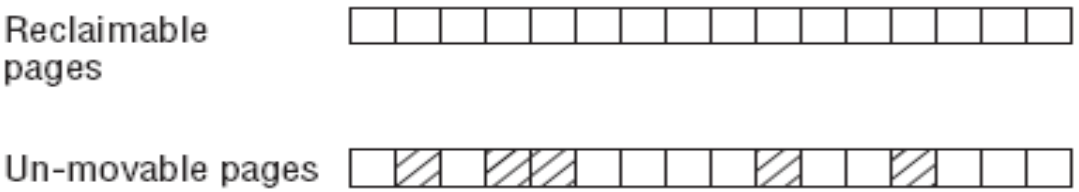


文件系统→碎片分析整理。
物理内存→许多物理内存页位置固定，不能移动。
anti-fragmentation，防止碎片。

Anti-fragmentation

- **Non-movable pages**
  - **have a fixed position in memory and can not be moved anywhere else.**
  - **Most allocations of the core kernel fall into this category.**
- **Reclaimable pages**
  - **Can not be moved directly, but they can be deleted and their contents regenerated from some source.**
  - **Data mapped from files fall into this category, for instance.**
  - **Periodically freed by the kswapd daemon depending on how often they are accessed.**
- **Movable pages**
  - **Can be moved around as desired.**
  - **Pages that belong to user space applications fall into this category. They are mapped via page tables.**

基于可移动性将页面分组



MIGRATE_TYPES

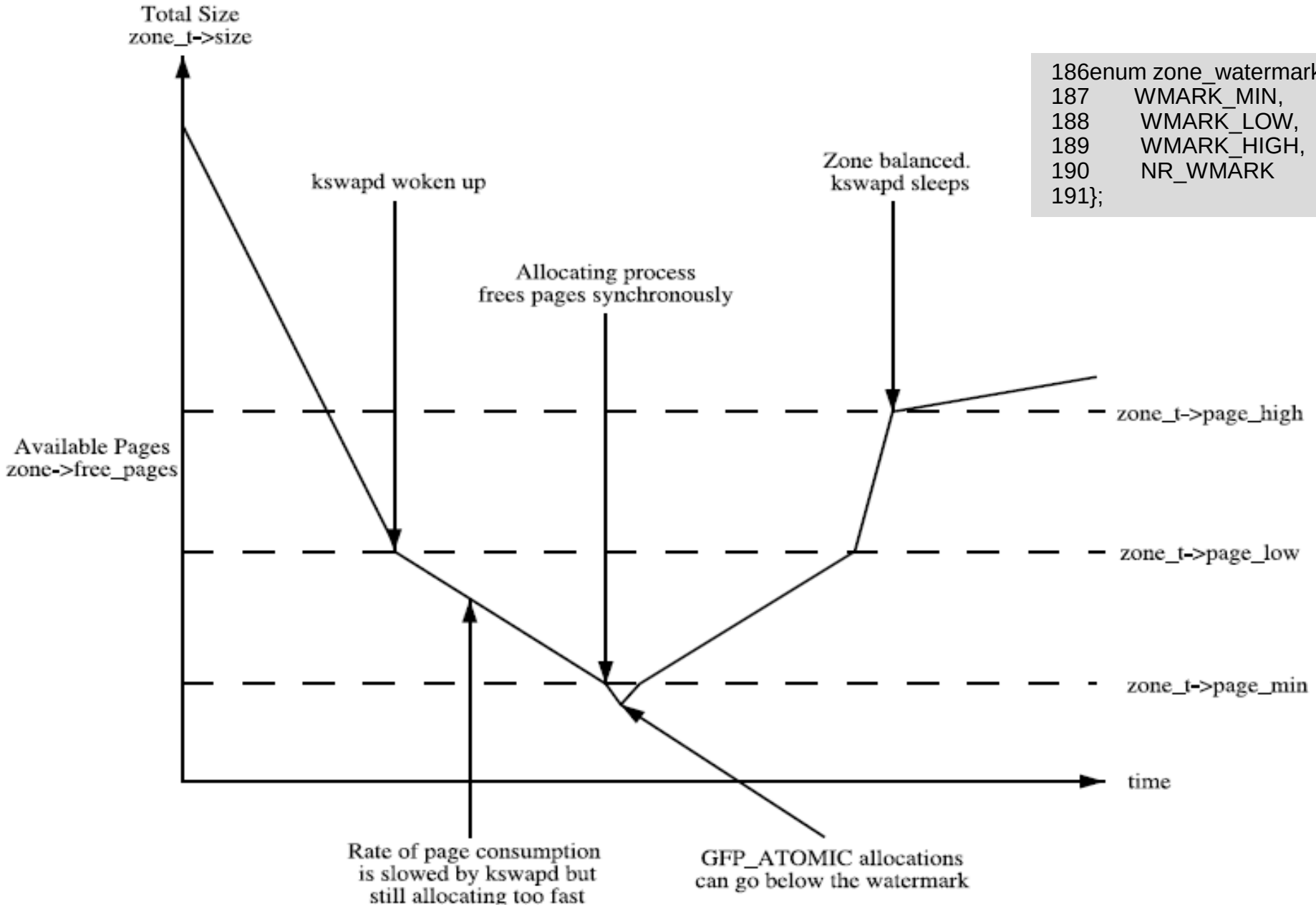
/include/linux/mmzone.h

```
enum {
    MIGRATE_UNMOVABLE, //核心分配的页大多数属于此类
    MIGRATE_MOVABLE, 2 //可随意移动，用户空间程序页属于此类
    MIGRATE_RECLAIMABLE, //不能直接移动，但是可以删除。Kswapd周期性释放此类内存
    MIGRATE_PCPTYPES,       /* the number of types on the pcp lists */
    MIGRATE_RESERVE = MIGRATE_PCPTYPES,
#ifdef CONFIG_CMA
    MIGRATE_CMA,
#endif
#ifdef CONFIG_MEMORY_ISOLATION
    MIGRATE_ISOLATE,        /* can't allocate from here */
#endif
    MIGRATE_TYPES   };
```

/mm/page_alloc.c

```
static int fallbacks[MIGRATE_TYPES][4] = {
    [MIGRATE_UNMOVABLE]   = { MIGRATE_RECLAIMABLE, MIGRATE_MOVABLE,   MIGRATE_TYPES },
    [MIGRATE_MOVABLE]     = { MIGRATE_RECLAIMABLE, MIGRATE_UNMOVABLE, MIGRATE_TYPES },
    [MIGRATE_RECLAIMABLE] = { MIGRATE_UNMOVABLE,   MIGRATE_MOVABLE,   MIGRATE_TYPES },
#ifdef CONFIG_CMA       [MIGRATE_CMA]       = { MIGRATE_TYPES }, /* Never used */       #endif
#ifdef CONFIG_MEMORY_ISOLATION    [MIGRATE_ISOLATE]    = { MIGRATE_TYPES }, /* Never used */    #endif  };
```

Zone Watermarks




```
186enum zone_watermarks {
187        WMARK_MIN,
188        WMARK_LOW,
189        WMARK_HIGH,
190        NR_WMARK
191};
```

alloc_pages

/include/linux/gfp.h

```
#ifdef CONFIG_NUMA
extern struct page *alloc_pages_current(gfp_t gfp_mask, unsigned
    order);
……
#else
#define alloc_pages(gfp_mask, order) \
     alloc_pages_node(numa_node_id(), gfp_mask, order)
#define alloc_pages_vma(gfp_mask, order, vma, addr, node, false)\
    alloc_pages(gfp_mask, order)
#define alloc_hugepage_vma(gfp_mask, vma, addr, order) \
    alloc_pages(gfp_mask, order)
#endif

#define alloc_page(gfp_mask) alloc_pages(gfp_mask, 0)
```

不考虑NUMA

```
static inline struct page *alloc_pages_node(int nid, gfp_t gfp_mask,
                                            unsigned int order)
{
    if (nid == NUMA_NO_NODE)
      nid = numa_mem_id();
    return __alloc_pages_node(nid, gfp_mask, order);   }

__alloc_pages_node(int nid, gfp_t gfp_mask, unsigned int order)
{
    VM_BUG_ON(nid < 0 || nid >= MAX_NUMNODES);
    VM_WARN_ON((gfp_mask & __GFP_THISNODE) && !node_online(nid));

    return __alloc_pages(gfp_mask, order, nid);
}

static inline struct page *
__alloc_pages(gfp_t gfp_mask, unsigned int order, int preferred_nid)
{
      return __alloc_pages_nodemask(gfp_mask, order, preferred_nid,
NULL);
}
```

```
__alloc_pages_nodemask(gfp_t gfp_mask, unsigned int order, int preferred_nid,
                                                        nodemask_t *nodemask)
{
    struct page *page;
    unsigned int alloc_flags = ALLOC_WMARK_LOW;
    gfp_t alloc_mask; /* The gfp_t that was actually used for allocation */
    struct alloc_context ac = { };

    if (unlikely(order >= MAX_ORDER)) {
        WARN_ON_ONCE(!(gfp_mask & __GFP_NOWARN));
        return NULL;
    }

    ……
    /* First allocation attempt */
    page = get_page_from_freelist(alloc_mask, order, alloc_flags, &ac);
    if (likely(page))
        goto out;
    ……
    page = __alloc_pages_slowpath(alloc_mask, order, &ac); //回收，降低水位线，
                                                          //再次调用快速分配
```

```
out:
        if (memcg_kmem_enabled() && (gfp_mask & __GFP_ACCOUNT) && page &&
            unlikely(__memcg_kmem_charge(page, gfp_mask, order) != 0)) {
                __free_pages(page, order);
                page = NULL;
        }

        trace_mm_page_alloc(page, order, alloc_mask, ac.migratetype);

        return page;
}
EXPORT_SYMBOL(__alloc_pages_nodemask);
```

```
/*goes through the zonelist trying to allocate a page. */
static struct page *
get_page_from_freelist(gfp_t gfp_mask, unsigned int order, int alloc_flags,
                                        const struct alloc_context *ac)
{
        struct zoneref *z;
        struct zone *zone;
        struct pglist_data *last_pgdat_dirty_limit = NULL;
        bool no_fallback;

retry:
        /*
         * Scan zonelist, looking for a zone with enough free.
         * See also __cpuset_node_allowed() comment in kernel/cpuset.c.
         */
        no_fallback = alloc_flags & ALLOC_NOFRAGMENT;
        z = ac->preferred_zoneref;
        for_next_zone_zonelist_nodemask(zone, z, ac->zonelist, ac->high_zoneidx,
                                                ac->nodemask) {
                ……
```

```
mark = wmark_pages(zone, alloc_flags & ALLOC_WMARK_MASK);
if (!zone_watermark_fast(zone, order, mark,
                    ac_classzone_idx(ac), alloc_flags)) {
    int ret;
    ……
    /* Checked here to keep the fast path fast */
    BUILD_BUG_ON(ALLOC_NO_WATERMARKS < NR_WMARK);
    if (alloc_flags & ALLOC_NO_WATERMARKS)
        goto try_this_zone;
    if (node_reclaim_mode == 0 ||
       !zone_allows_reclaim(ac->preferred_zoneref->zone, zone))
        continue;
    ret = node_reclaim(zone->zone_pgdat, gfp_mask, order); //快速回收
    switch (ret) {
    case NODE_RECLAIM_NOSCAN:/* did not scan */
        continue;
    case NODE_RECLAIM_FULL: /* scanned but unreclaimable */
        continue;
    default:  /* did we reclaim enough */
        if (zone_watermark_ok(zone, order, mark,
                    ac_classzone_idx(ac), alloc_flags))
            goto try_this_zone;
        continue;
    }
}
```

```
try_this_zone:
        page = rmqueue(ac->preferred_zoneref->zone, zone, order,
                    gfp_mask, alloc_flags, ac->migratetype);
        if (page) {
            prep_new_page(page, order, gfp_mask, alloc_flags);

            /*
             * If this is a high-order atomic allocation then check
             * if the pageblock should be reserved for the future
             */
            if (unlikely(order && (alloc_flags & ALLOC_HARDER)))
                reserve_highatomic_pageblock(page, zone, order);

            return page;
        } else {        ……     }
    }

    if (no_fallback) {
        alloc_flags &= ~ALLOC_NOFRAGMENT;
        goto retry;
    }

    return NULL;
}
```

```
/*
 * Allocate a page from the given zone. Use pcplists for order-0 allocations.
 */
static inline
struct page *rmqueue(struct zone *preferred_zone,
                        struct zone *zone, unsigned int order,
                        gfp_t gfp_flags, unsigned int alloc_flags,
                        int migratetype)
{
        unsigned long flags;
        struct page *page;

        if (likely(order == 0)) {
                page = rmqueue_pcplist(preferred_zone, zone, gfp_flags,
                                        migratetype, alloc_flags);
                goto out;
        }
        /*
         * We most definitely don't want callers attempting to
         * allocate greater than order-1 page units with __GFP_NOFAIL.
         */
        WARN_ON_ONCE((gfp_flags & __GFP_NOFAIL) && (order > 1));
```

```
spin_lock_irqsave(&zone->lock, flags);
do {
	page = NULL;
	if (alloc_flags & ALLOC_HARDER) {
		page = __rmqueue_smallest(zone, order, MIGRATE_HIGHATOMIC);
		if (page)
			trace_mm_page_alloc_zone_locked(page, order, migratetype);
	}
	if (!page)
		page = __rmqueue(zone, order, migratetype, alloc_flags);
} while (page && check_new_pages(page, order));
spin_unlock(&zone->lock);
if (!page)
	goto failed;
__mod_zone_freepage_state(zone, -(1 << order),
			  get_pcppage_migratetype(page));

__count_zid_vm_events(PGALLOC, page_zonenum(page), 1 << order);
zone_statistics(preferred_zone, zone);
local_irq_restore(flags);
```

```
out:
    /* Separate test+clear to avoid unnecessary atomics */
    if (test_bit(ZONE_BOOSTED_WATERMARK, &zone->flags)) {
        clear_bit(ZONE_BOOSTED_WATERMARK, &zone->flags);
        wakeup_kswapd(zone, 0, 0, zone_idx(zone));
    }

    VM_BUG_ON_PAGE(page && bad_range(zone, page), page);
    return page;

failed:
    local_irq_restore(flags);
    return NULL;
}
```

```
/* * Do the hard work of removing an element from the buddy allocator.
 * Call me with the zone->lock already held. */
static __always_inline struct page *
__rmqueue(struct zone *zone, unsigned int order, int migratetype,
                                    unsigned int alloc_flags)
{
        struct page *page;

retry:
        page = __rmqueue_smallest(zone, order, migratetype);
        if (unlikely(!page)) {
                if (migratetype == MIGRATE_MOVABLE)
                        page = __rmqueue_cma_fallback(zone, order);

                if (!page && __rmqueue_fallback(zone, order, migratetype,
                                                alloc_flags))
                        goto retry;
        }

        trace_mm_page_alloc_zone_locked(page, order, migratetype);
        return page;
}
```

```
/* * Go through the free lists for the given migratetype and remove
 * the smallest available page from the freelists*/
static __always_inline
struct page *__rmqueue_smallest(struct zone *zone, unsigned int order,
                                int migratetype)
{
        unsigned int current_order;
        struct free_area *area;
        struct page *page;

        /* Find a page of the appropriate size in the preferred list */
        for (current_order = order; current_order < MAX_ORDER; ++current_order) {
                area = &(zone->free_area[current_order]);
                page = get_page_from_free_area(area, migratetype);
                if (!page)
                        continue;
                del_page_from_free_area(page, area);
                expand(zone, page, order, current_order, area, migratetype);
                set_pcppage_migratetype(page, migratetype);
                return page;
        }

        return NULL;
}
```

```
static inline void expand(struct zone *zone, struct page *page,
     int low, int high, struct free_area *area,
     int migratetype)
{
     unsigned long size = 1 << high;

     while (high > low) {
          area--;
          high--;
          size >>= 1;
          VM_BUG_ON_PAGE(bad_range(zone, &page[size]), &page[size]);

          /*
           * Mark as guard pages (or page), that will allow to
           * merge back to allocator when buddy will be freed.
           * Corresponding page table entries will not be touched,
           * pages will stay not present in virtual address space
           */
          if (set_page_guard(zone, &page[size], high, migratetype))
               continue;

          add_to_free_area(&page[size], area, migratetype);
          set_page_order(&page[size], high);
     }
}
```

如果将要分配的空闲内存块的order和比请求的 order 大，说明 buddy Allocator 分配的内存块是从更大的空闲内存块得到的

代码的思想：当需要2^k 个page frames的block来满足分配需要2^h (h < k)的时候， 程序分配第一个 2^h page frames，然后循环把 $2^k - 2^h$ page frames 加入到相应 order 介于 h 与 k 的 free_area lists。