Linux Memory Management

referenced kernel：2.6.10, 2.6.34

Guanghua Song, Zhejiang University

ghsong@zju.edu.cn

Dec., 2018

About assembly code in C

```
/* include/asm-i386/string.h */
static inline void * __memcpy(void * to, const void * from, size_t n)
{
int d0, d1, d2;
__asm__ __volatile__(
        "rep ; movsl\n\t"
        "testb $2,%b4\n\t"
        "je 1f\n\t"
        "movsw\n"
        "1:\ttestb $1,%b4\n\t"
        "je 2f\n\t"
        "movsb\n"
        "2:"
        : "=&c" (d0), "=&D" (d1), "=&S" (d2)
        :"0" (n/4), "q" (n),"1" ((long) to),"2" ((long) from)
        : "memory");
return (to);
}
```

指令部
:输出部
: 输入部
: 损坏部

关于中断（interrupt）

- 硬中断
  - 外部设备引起，也称外部中断
  - 异步性
- 软中断
  - INT n，由CPU执行指令引起，如int 0x80
  - 也称trap（陷阱）
- 异常（exception）
  - 如执行除法指令时除数为0，或缺页page fault等
- 发生中断时通过中断向量表(IDT)获得入口，但IDT表项已经从传统的入口地址变成了更复杂的描述项，称为门(gate)，符合准入条件才能进入
  - interrupt gate, trap gate, task gate, call gate

中断处理要求

- INTEL体系结构对中断处理的要求
  - 特权级：Privilege Level, 0～3, 值越小，特权级越高(Linux、Windows只使用0和3)
  - 总体原则：当前代码段的特权级（CPL, Current Privilege Level）不高于中断处理程序所在段的特权级（DPL, Descriptor Privilege Level），即CPL≥中断服务程序的DPL
  - 对软中断，要求CPL≤中断或陷阱门的DPL，即：用户程序不能随便通过软中断进入更高特权级的中断处理程序；但Linux对向量3,4,5和0x80不检查中断门的DPL，因此，用户程序可以发起*int3（断点）, into（溢出）, bound（数组边界）, int 0x80（系统调用）*这4个软中断（硬中断不做此检查）
  - 特权级为0的中断处理程序必须使用相应特权级的堆栈（当前进程的系统堆栈），所以从用户程序进入中断处理程序必须进行堆栈切换。

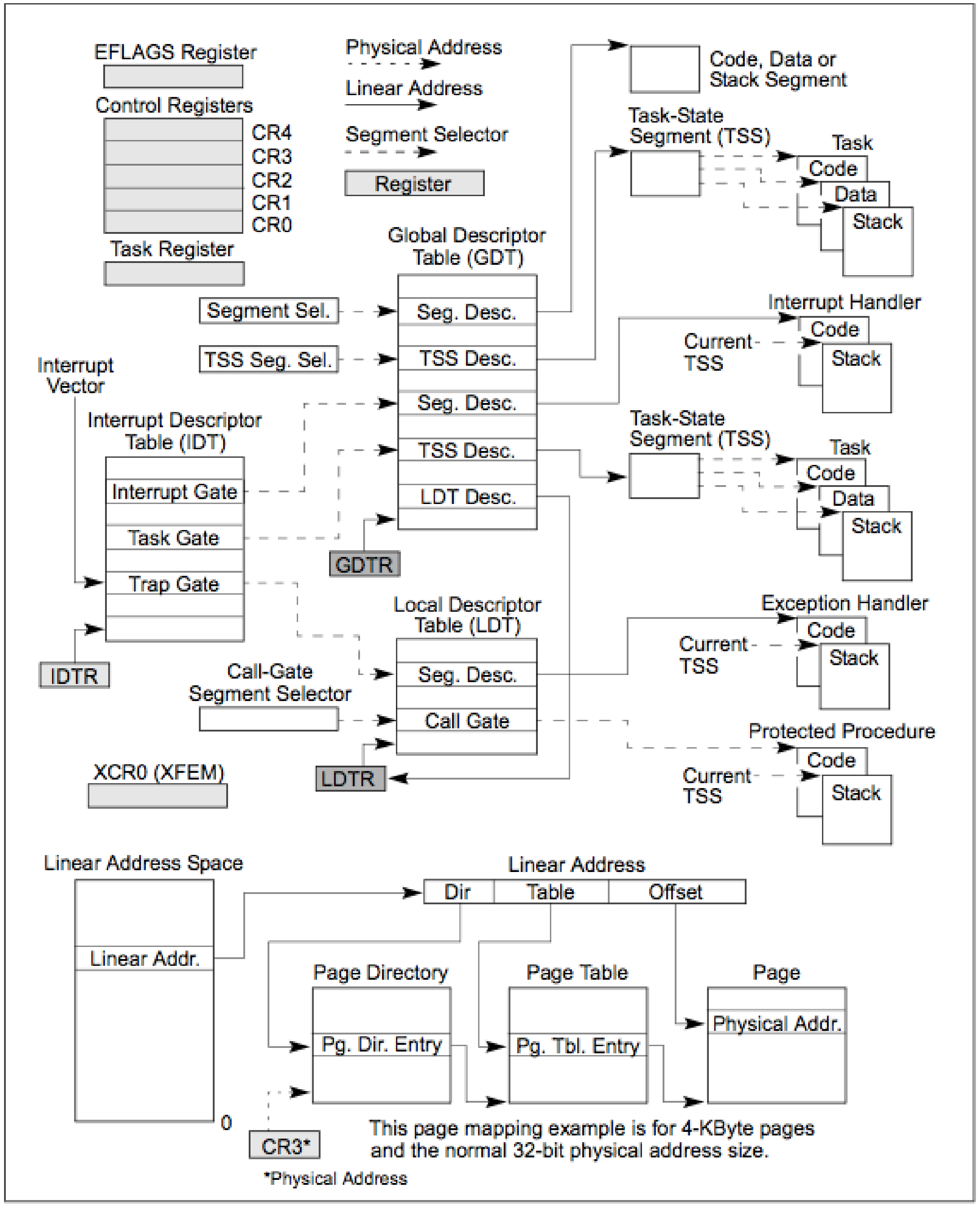


Figure 2-1. IA-32 System-Level Registers and Data Structures

- IDT: 1，中断描述符表
- GDT: 1，全局描述符表，本身不是一个段
- LDT: 0/1/n，局部描述符表，本身是一个段

访问数据段时特权级检查

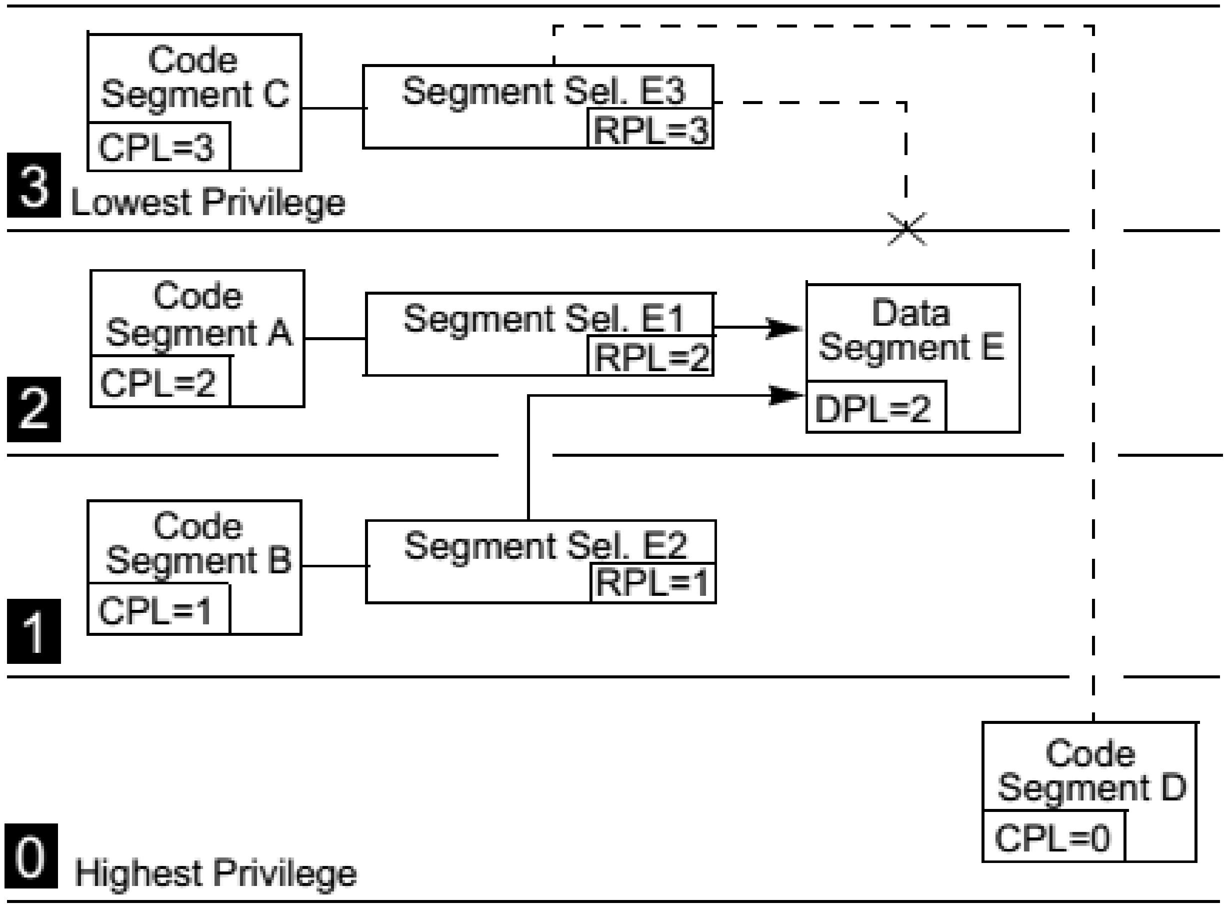


Figure 5-5. Examples of Accessing Data Segments From Various Privilege Levels

代码段切换时特权级检查

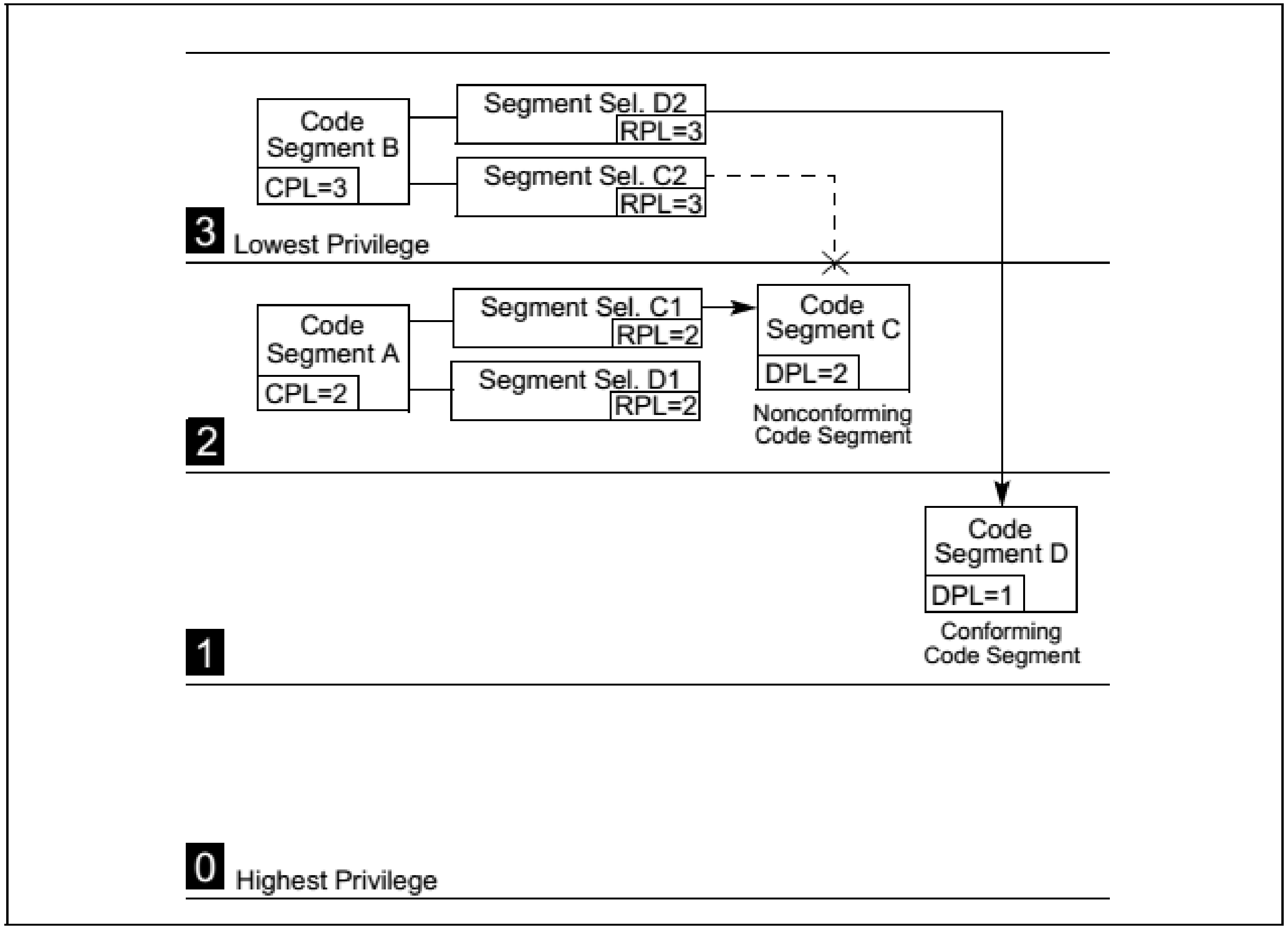


Figure 5-7. Examples of Accessing Conforming and Nonconforming Code Segments From Various Privilege Levels

通过中断访问代码段

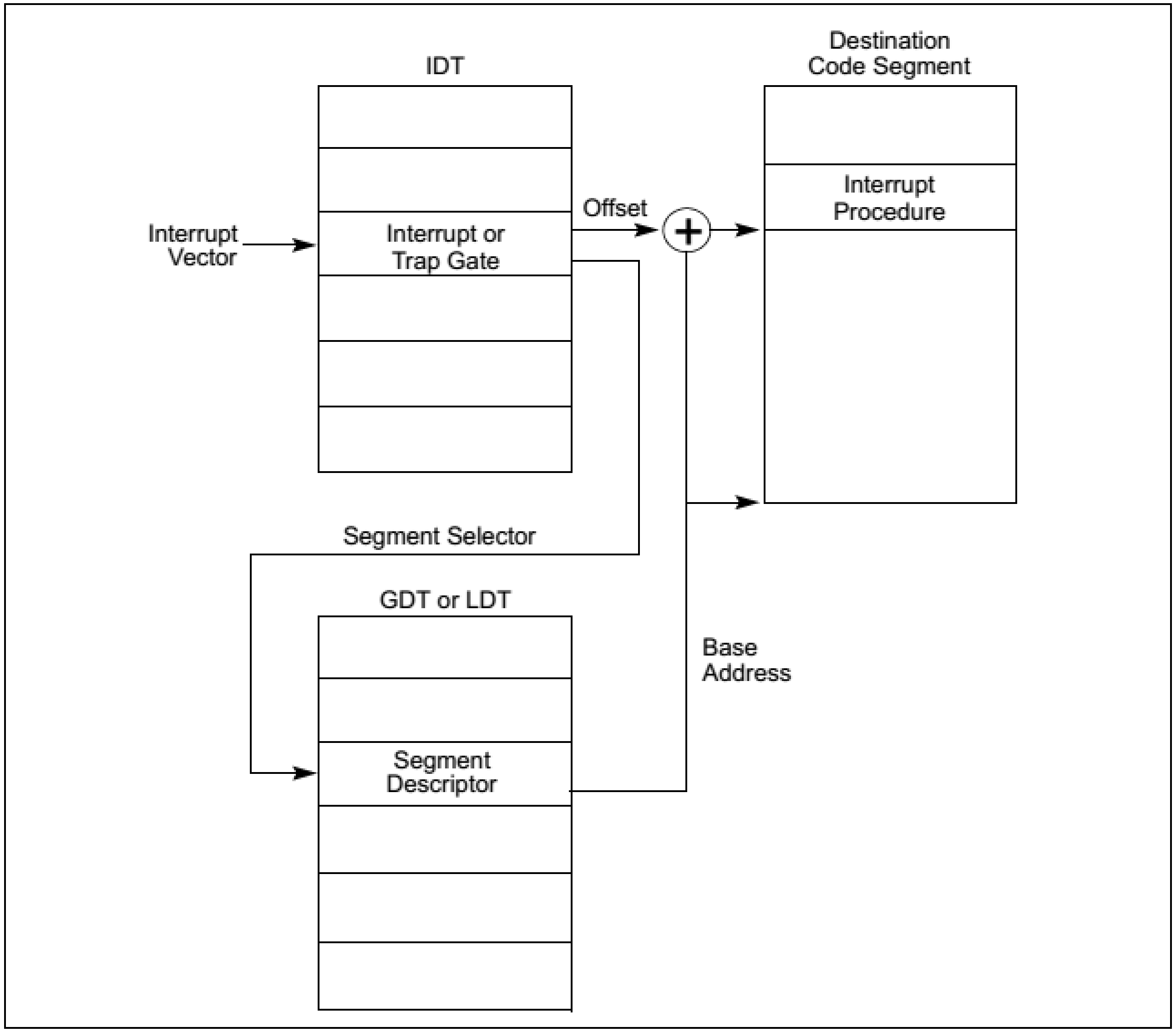


Figure 6-3. Interrupt Procedure Call

中断前后堆栈切换

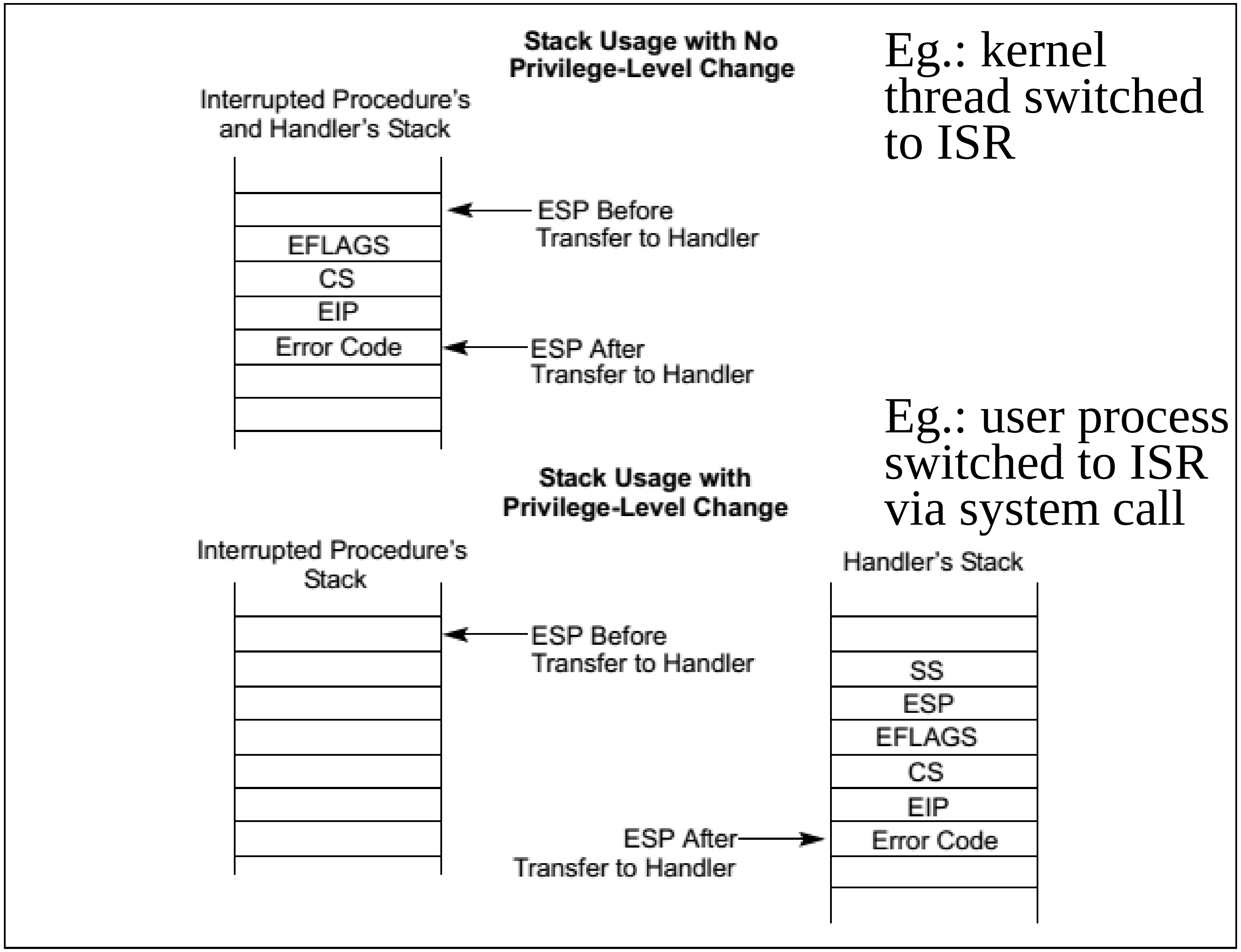

Stack Usage with No Privilege-Level Change
Interrupted Procedure's and Handler's Stack
EFLAGS
CS
EIP
Error Code
ESP Before Transfer to Handler
ESP After Transfer to Handler
Eg.: kernel thread switched to ISR
Stack Usage with Privilege-Level Change
Eg.: user process switched to ISR via system call
Interrupted Procedure's Stack
Handler's Stack
ESP Before Transfer to Handler
SS
ESP
EFLAGS
CS
EIP
Error Code
ESP After Transfer to Handler

Figure 6-4. Stack Usage on Transfers to Interrupt and Exception-Handling Routines

Error Code: if exception causes error

再看system call

- Linux 用户程序通过int 0x80发起系统调用(window为int 0x2e)
- 即软中断，发生特权级变化，所以应进行堆栈切换，因此，利用寄存器而非堆栈传递系统调用参数
  - 发起int 0x80前，通过寄存器传参数(包括系统调用号和参数)

  系统调用sethostname

  库函数int sethostname(const char *name, size_t len):

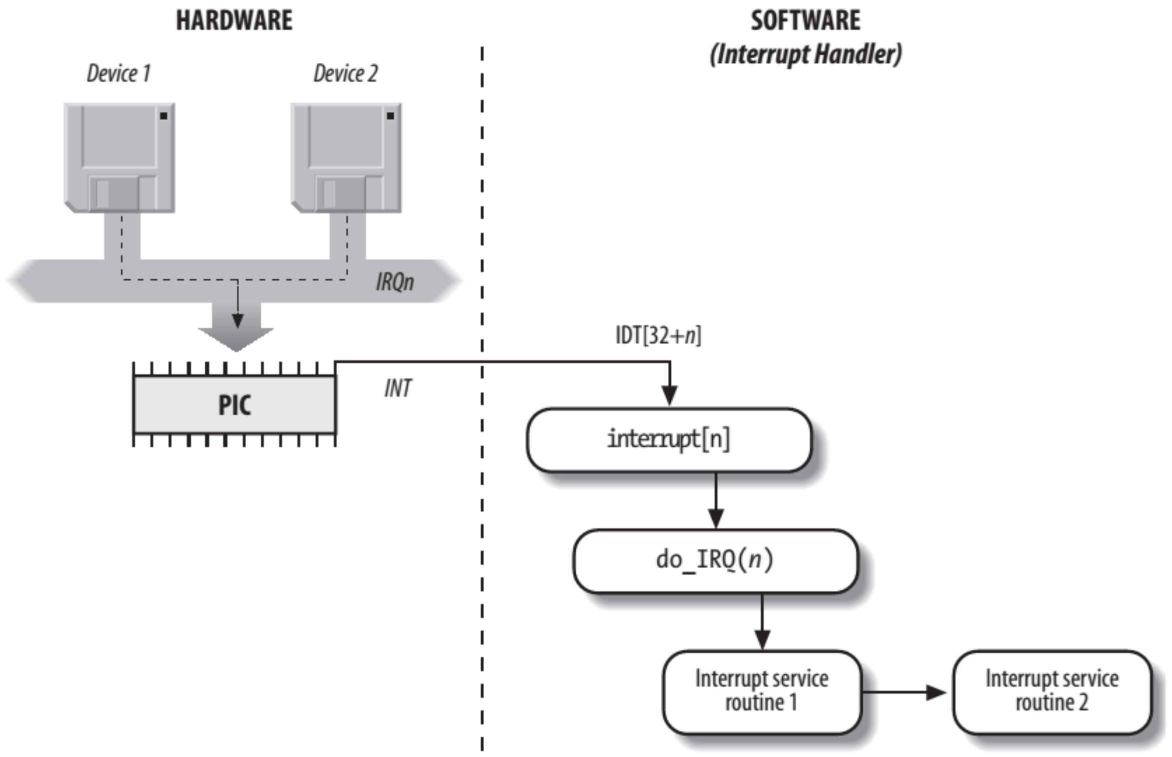
  库函数中如何发起系统调用:

```
movl 0x8(%esp,1),%ecx     //参数len传入寄存器ecx
movl 0x4(%esp,1),%ebx     //参数name传入寄存器ebx
movl $0x4a, %eax   //系统调用号传入寄存器eax
int $0x80          //发起系统调用
…
ret                //库函数返回
```

Linux中断处理

- I/O interrupt handling

1. Save the IRQ value and the register’s contents on the Kernel Mode stack.
2. Send an acknowledgment to the PIC that is servicing the IRQ line, thus allowing it to issue further interrupts.
3. Execute the interrupt service routines (ISRs) associated with all the devices that share the IRQ.
4. Terminate by jumping to the ret_from_intr( )address.



Linux中断处理

- Timer interrupt
  1. Keeping the current time and date so they can be returned to user programs through the time( ), ftime( ), and gettimeofday( ) APIs and used by the kernel itself as timestamps for files and network packets.
  2. Maintaining timers—mechanisms that are able to notify the kernel or a user program that a certain interval of time has elapsed.
  3. performing periodic work.

     ① Update resource usages, such as consumed system and user time, for the currently running process.

     ② Run any dynamic timers that have expired.

     ③ Execute scheduler_tick().

     ④ Update the wall time, which is stored in xtime.

     ⑤ Calculate load average.

进程上下文与中断上下文

- CPU在任何时刻处于以下三种情况之一

1. 运行于用户空间，执行用户进程，处于进程上下文**(process context)**
2. 运行于内核空间，处于进程（一般是**kernel thread**，内核线程）上下文
3. 运行于内核空间，处于中断（中断服务程序**ISR**，包括系统调用处理过程）上下文**(interrupt context)**

- 内核线程**kernel thread**以进程上下文的形式运行在内核空间中，本质上还是进程，但它有调用内核代码的权限，比如主动调用**schedule()**函数出让**CPU**等
- **3.0**版本前，内核线程主要是一些周期性的后台任务，如**kswapd**等，**3.0**版本后，原来中断服务程序的后半部（**bottom half**）都统一由内核线程来处理。

Memory Management

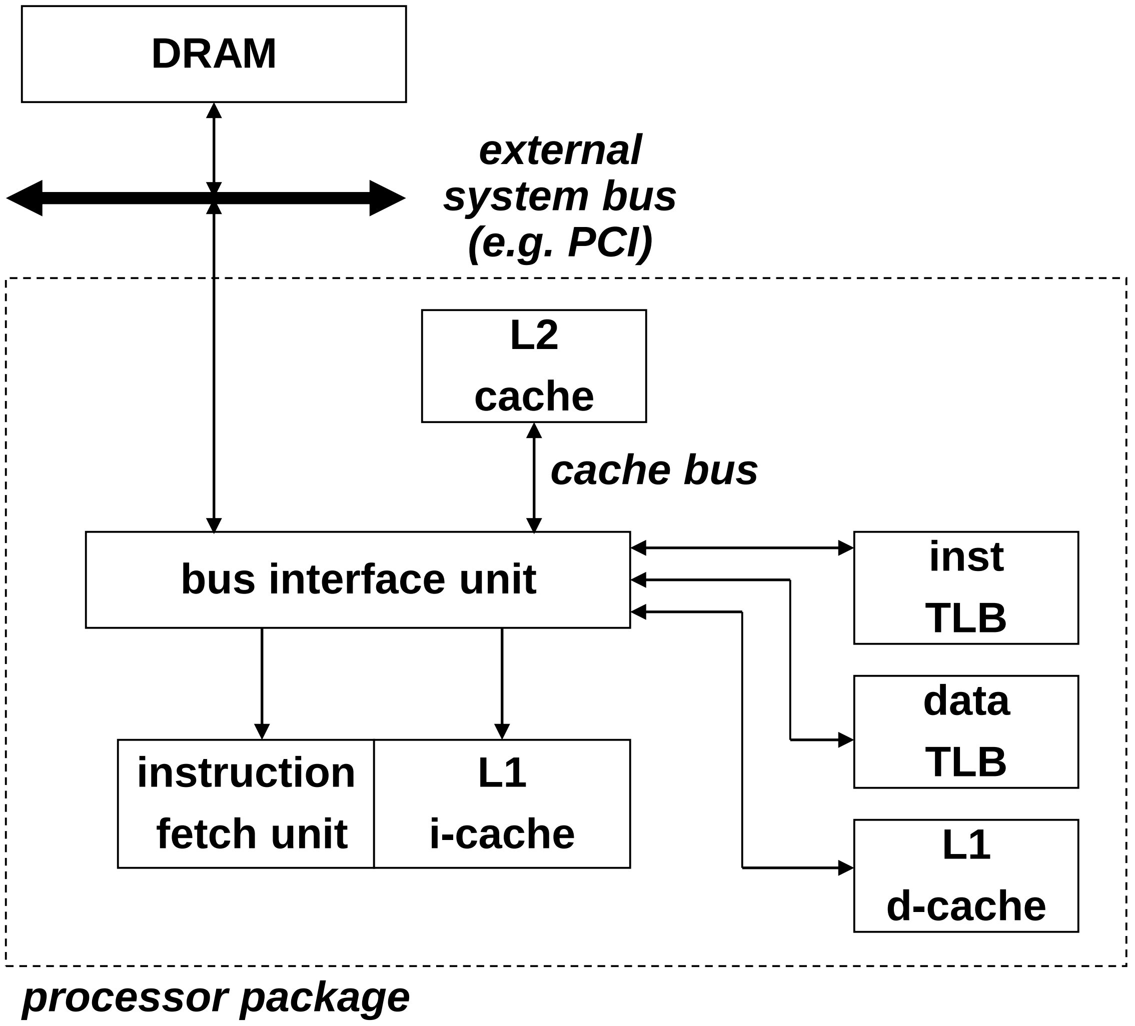
- Topics
  - P6 address translation
  - Physical memory management
  - Virtual memory management
  - Page fault handling
  - Memory mapping
  - Kernel memory management

Intel P6(IA-32架构, 1995, 0.6微米工艺)

P6、NetBurst(Pentium 4, D,M)、Core、Nehalem(i7,Intel 64)、Sandy Bridge(Xeon E5)

- Internal Designation for Successor to Pentium(586)
  - 550 transistors, 133 MHz
- Fundamentally Different from Pentium
  - Superscalar operation, Designed to handle server applications
- Resulting Processors
  - PentiumPro (1996), Pentium II (1997)
    - ✓ Incorporated MMX (MultiMedia eXtensions) instructions
      - special instructions for parallel processing
    - ✓ L2 cache on chip
  - Pentium III (1999)
    - ✓ Incorporated Streaming SIMD Extensions
      - More instructions for parallel processing
- IA-32 architecture is the instruction set architecture and programming environment for Intel's 32-bit microprocessors.
- Intel® 64 architecture is the instruction set architecture and programming environment which is a superset of and compatible with IA-32 architecture.

P6 Memory System



TLB: Translation Look-aside Buffer

32 bit address space

4 KB page size

inst TLB

- 4 ways
- 32 entries
- 8 sets

data TLB

- 4 ways
- 64 entries
- 16 sets

L1 i-cache and d-cache

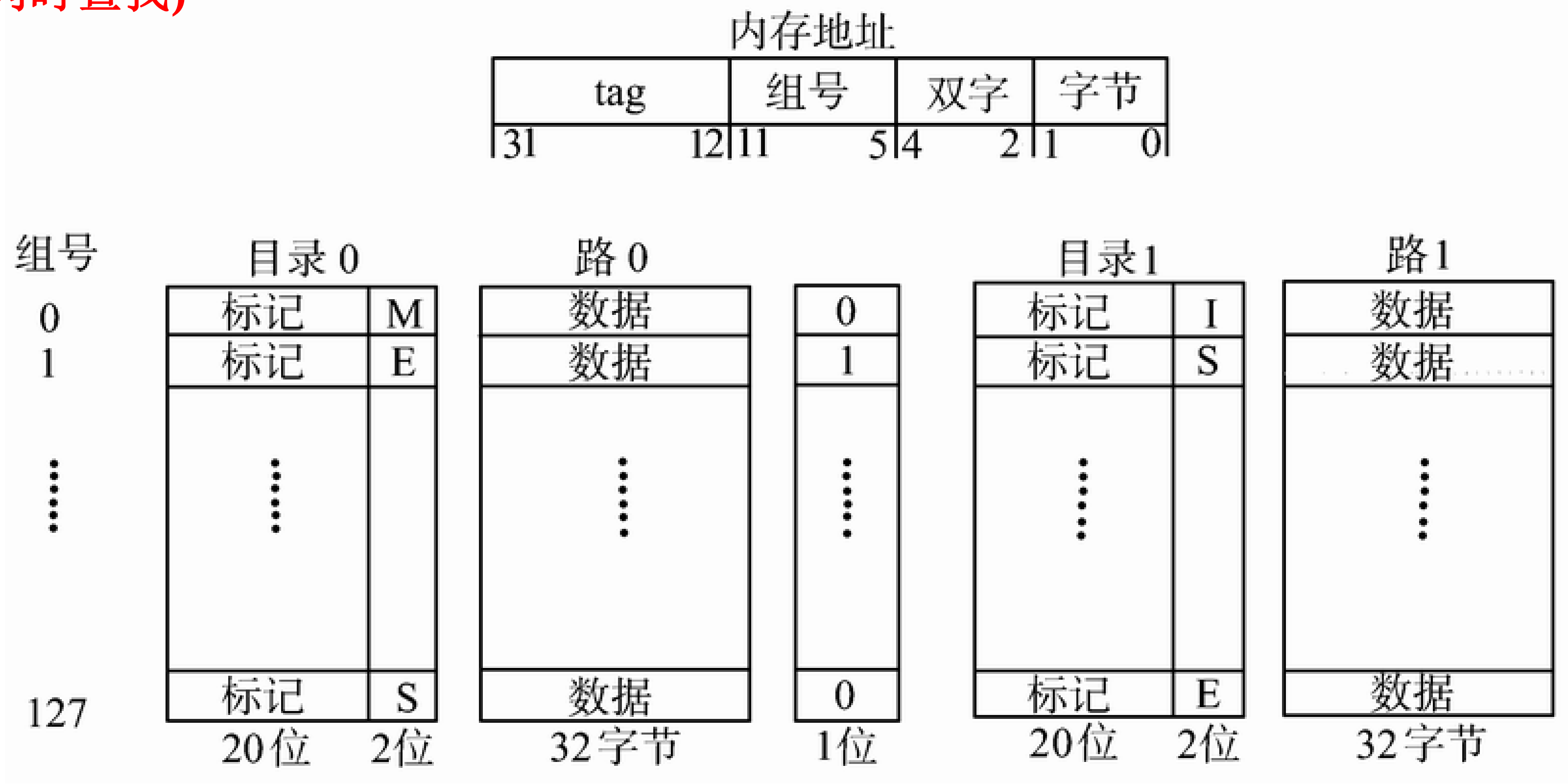
- 2 ways each
- 8KB each
- 32 B line size
- 128 sets

L2 cache

- unified
- 128 KB -- 2 MB

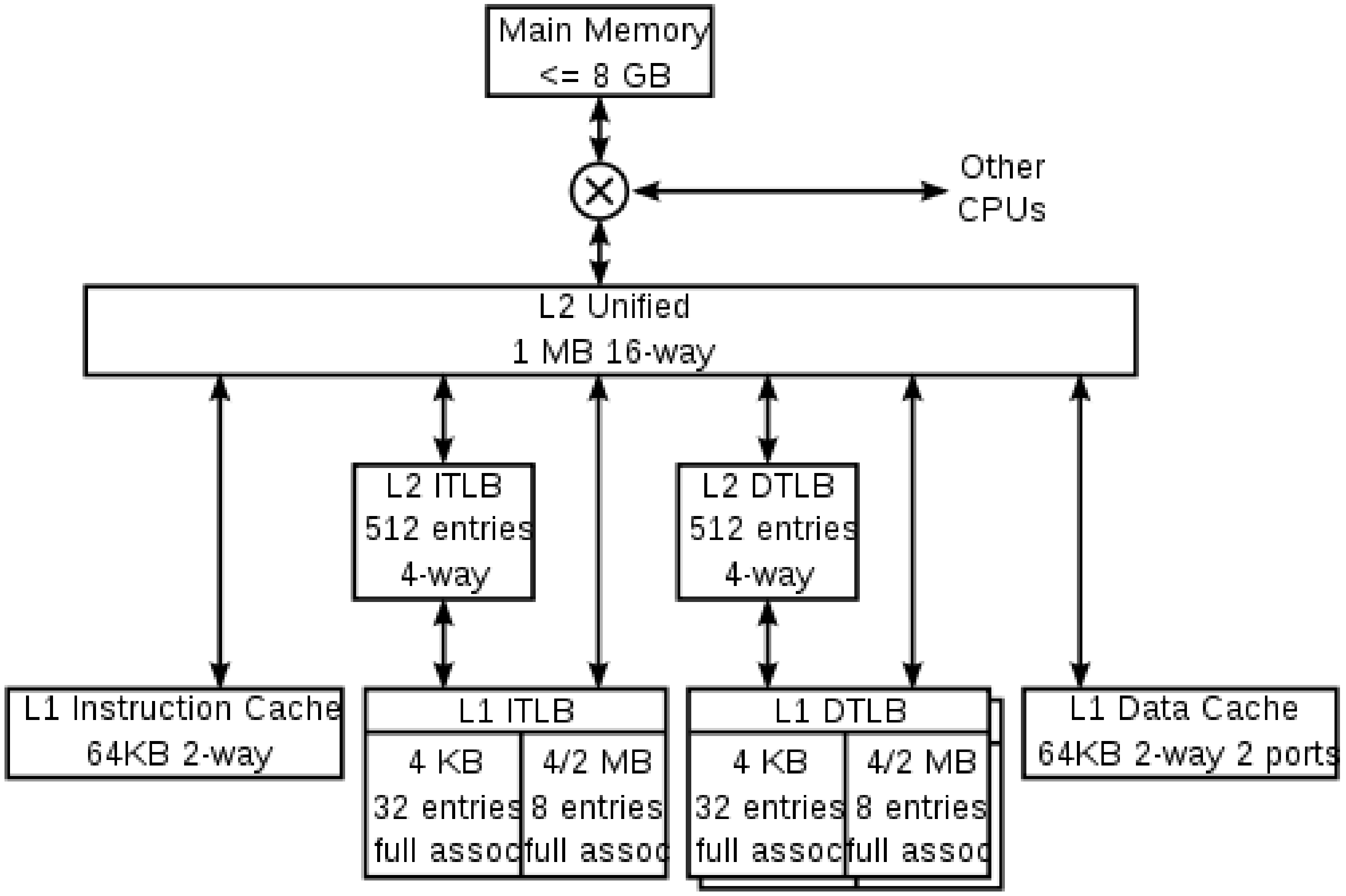
P6 L1 data cache

8KB的数据cache采用两路组相联(2-way set associative)的组织方式，分成128组，每组2 ways（路），每路32字节 （称为cache line size）。每路有一个20位的标记和2位的M / E / S / I 的状态位，这22位构成该路的目录项。采用LRU替换策略，一组两路共用一个LRU位。L1指令cache：类同于此，但2位状态位被替换为1位有效位。(多路同时查找)



M(Modified), E(Exclusive), S(Shared), I(Invalid)

AMD Athlon 64 CPU (x86-64)



Cache hierarchy of the K8 core in the AMD Athlon 64 CPU

i7处理器：L2: 256KB每核，L3: 4-20MB共享（典型8MB）

Intel Pentium 4 and Xeon

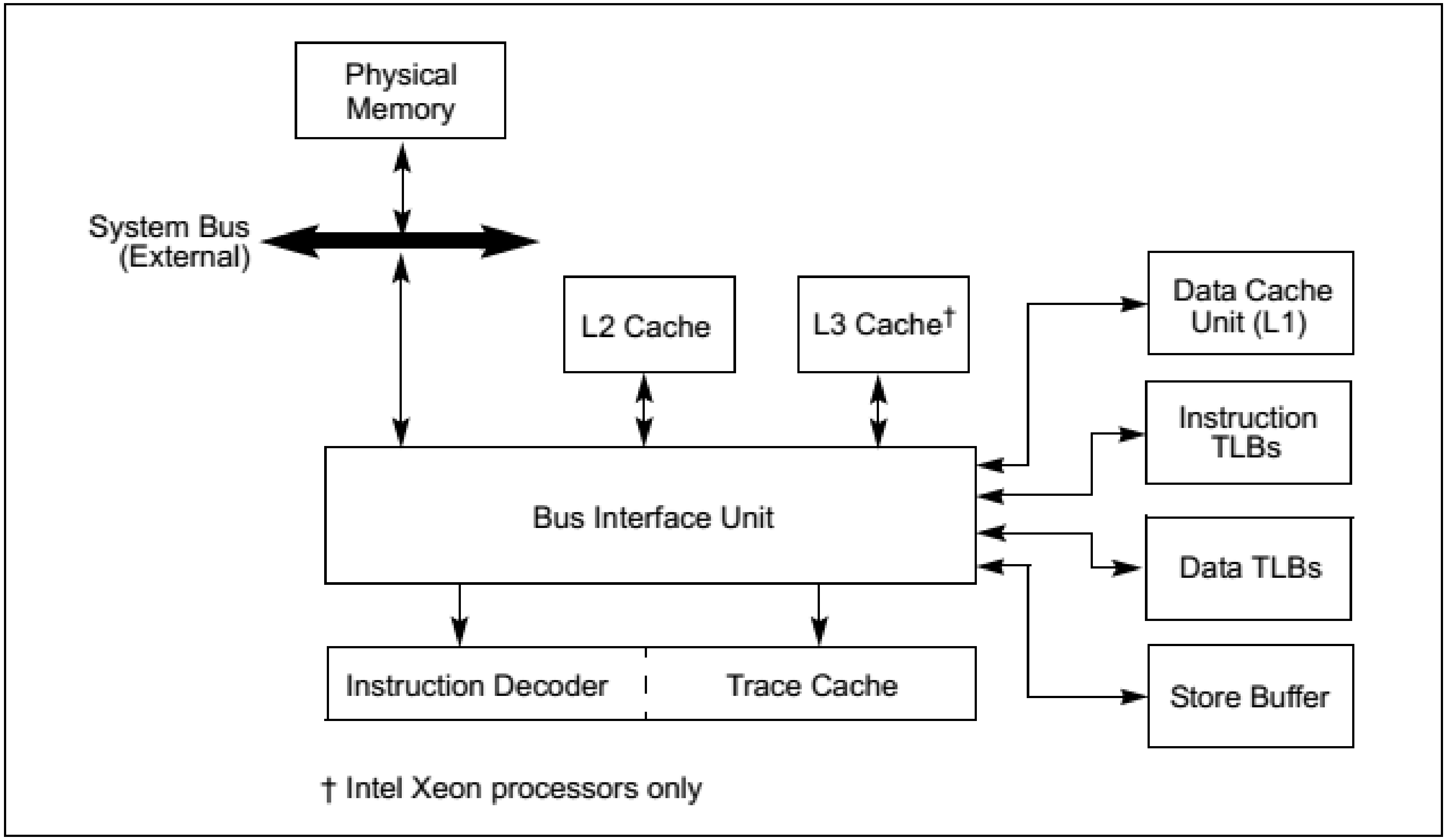


Figure 11-1. Cache Structure of the Pentium 4 and Intel Xeon Processors

Segmentation and Paging

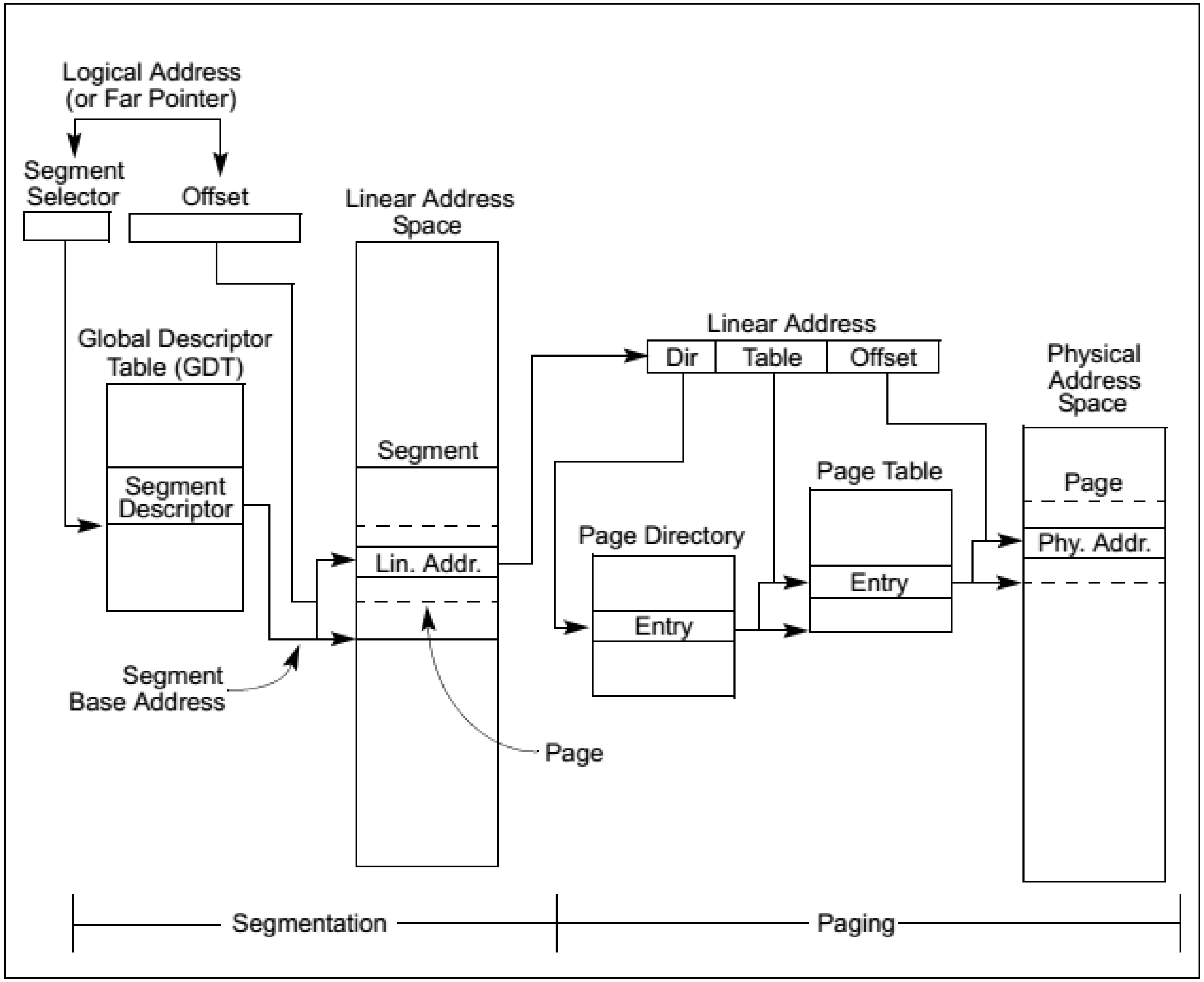
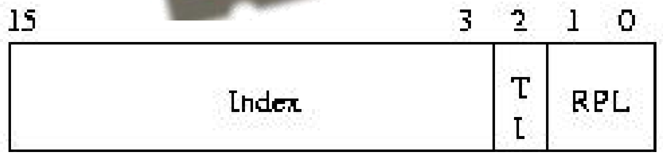


Figure 3-1. Segmentation and Paging

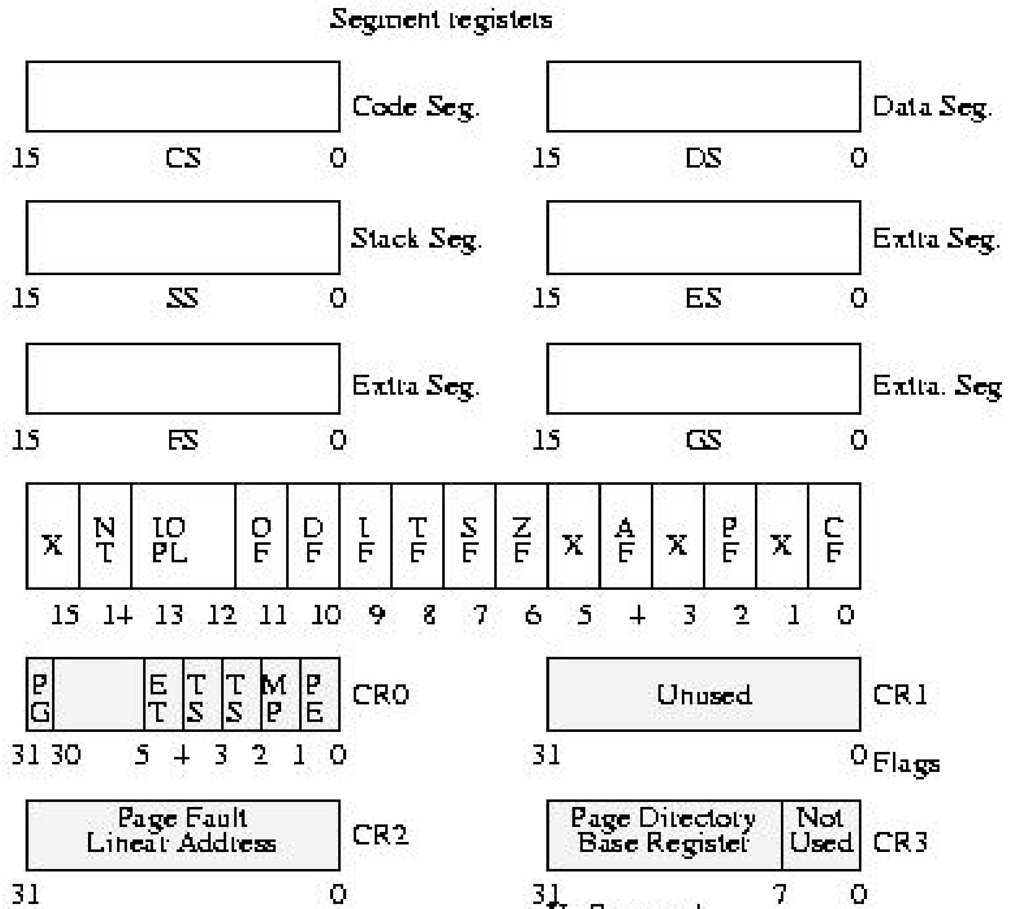
Intel x86 Special Registers



RPL = Requestor Privilege Level
TI = Table Indicator
(0 = GDT, 1 = LDT)
Index = Index into table

Protected Mode segment selector

Typical Segment Register
Current Priority is RPL
Of Code Segment (CS)



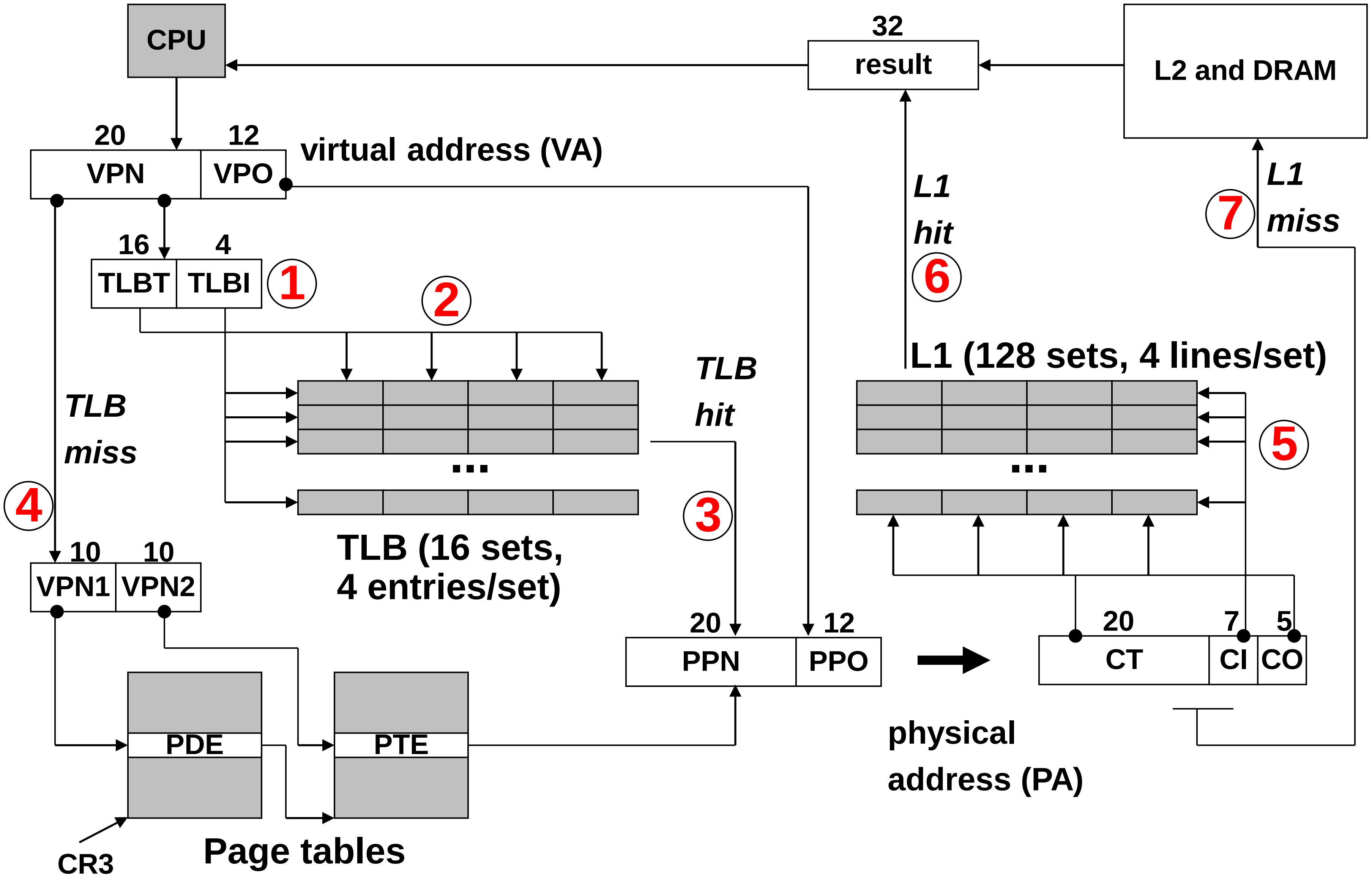
PG=Paging Enable
ET=Emulation Type
TS=Task Switched
EM=Emulate Coprocessor
MP=Math coprocessor present
PE=Protected Mode enable

X=Reserved
NT=Nested Task
IOPL=I/O Privilege Level
OF=Overflow Flag
DF=Direction Flag
IF=Interrupt Flag
TF=Trap Flag
SF=Sign Flag
ZF=Zero Flag
AF=Auxiliary Flag
PF=Parity Flag
CF=Carry Flag

P6 Address Translation - Abbreviations

- Symbols:
  - Components of the virtual address (VA)
    - ✓TLBI: TLB index
    - ✓TLBT: TLB tag
    - ✓VPO: virtual page offset
    - ✓VPN: virtual page number
  - Components of the physical address (PA)
    - ✓PPO: physical page offset (same as VPO)
    - ✓PPN: physical page number
    - ✓CO: byte offset within cache line
    - ✓CI: cache index
    - ✓CT: cache tag

Overview of P6 Address Translation

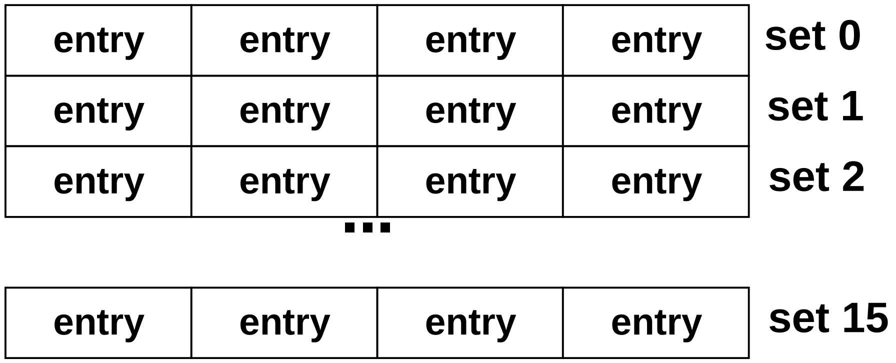


P6 TLB

- TLB entry:

32	16	1	1
PDE/PTE	Tag	PD	V

  - V: indicates a valid (1) or invalid (0) TLB entry
  - PD: is this entry a PDE (1) or a PTE (0)?
  - tag: disambiguates entries cached in the same set
  - PDE/PTE: page directory or page table entry
- Structure of the data TLB:
  - 16 sets, 4 entries/set



P6 Page Directory Entry (PDE)

31 12	11 9	8	7	6	5	4	3	2	1	0
Page table physical base addr	Avail	G	PS		A	CD	WT	U/S	R/W	P=1

Page table physical base address: 20 most significant bits of physical page table address (forces page tables to be 4KB aligned)

Avail: These bits available for system programmers

G: global page (don't evict from TLB on task switch)

PS: page size 4K (0) or 4M (1)

A: accessed (set by MMU on reads and writes, cleared by software)

CD: cache disabled (1) or enabled (0)

WT: write-through or write-back cache policy for this page table

U/S: user or supervisor mode access

R/W: read-only or read-write access

P: page table is present in memory (1) or not (0)

31 1	0
Available for OS (page table location in secondary storage)	P=0

P6 Page Table Entry (PTE)

31 12	11 9	8	7	6	5	4	3	2	1	0
Page physical base address	Avail	G	0	D	A	CD	WT	U/S	R/W	P=1

Page base address: 20 most significant bits of physical page address (forces pages to be 4 KB aligned)

Avail: available for system programmers

G: global page (don’t evict from TLB on task switch)

D: dirty (set by MMU on writes)

A: accessed (set by MMU on reads and writes)

CD: cache disabled or enabled

WT: write-through or write-back cache policy for this page

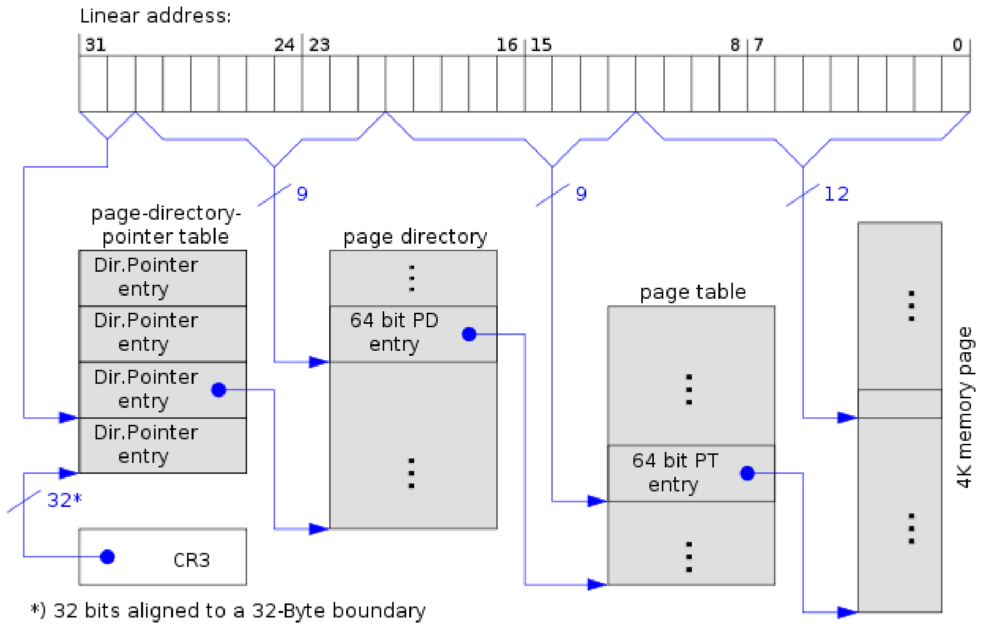
U/S: user/supervisor

R/W: read/write

P: page is present in physical memory (1) or not (0)

31 1	0
Available for OS (page location in secondary storage)	P=0

Slightly More than 4GB RAM: PAE mode on x86 (Since Pentium Pro)

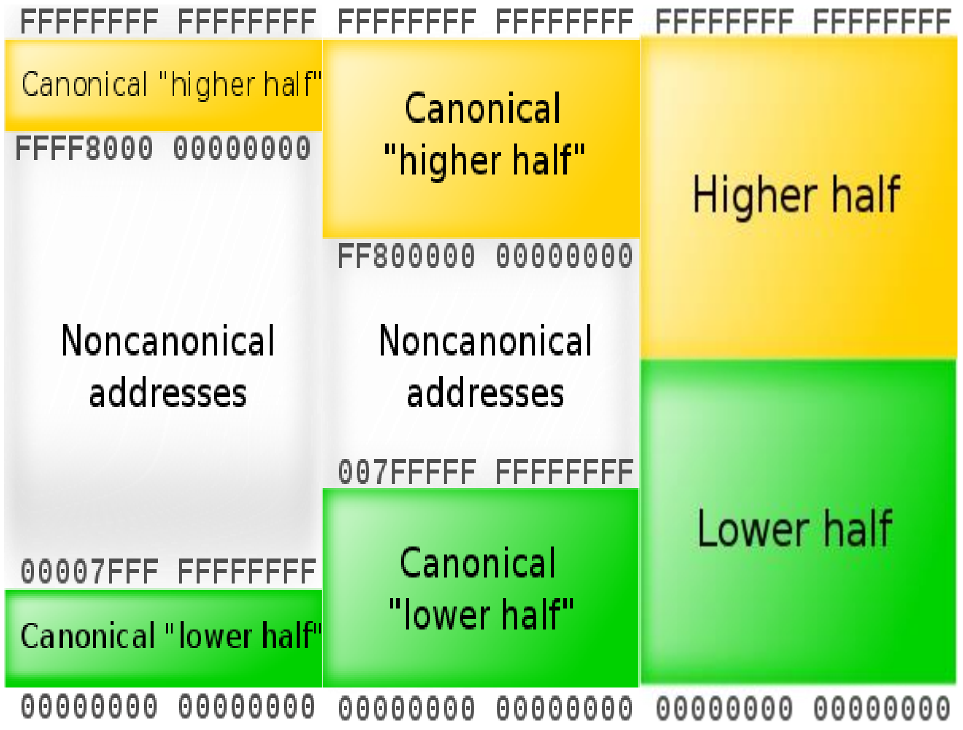


This forces kernel programmers to reuse the same linear addresses to map different areas of RAM.用户空间仍为4GB, OS内核使用4GB以上的内存

- Physical Address Extension (PAE)
  - Poor-man's large memory extensions
  - More than 4GB physical memory (36~52 memory address pins)
  - Every process still can have only 32-bit address space
- 3-Level page table
  - 64-bit PTE format
- How do processes use more than 4GB memory?
  - OS Supports for mapping and unmapping physical memory into virtual address space
  - Address Windowing Extensions (AWE) （超过4GB的应用程序）

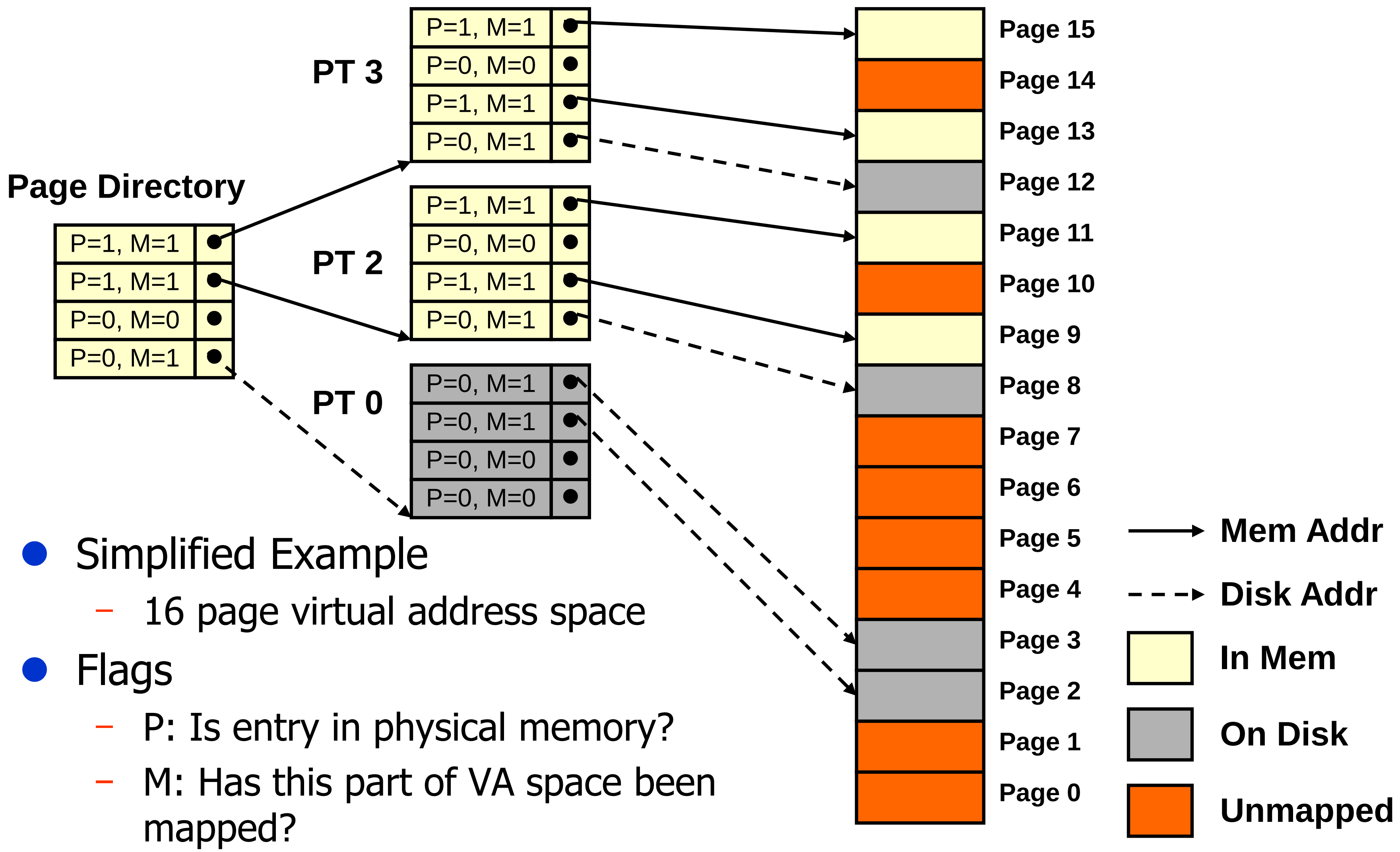
What about 64-bit x86-64 (“long mode”)?

- X86 long mode: 64 bit virtual addresses, 40-52 bits of physical memory
  - Not all 64-bit virtual addresses translated
  - Virtual Addresses must be “cannonical”(规范): top n bits must be equal
    - ✓ n here might be 64-48
    - ✓ Non-cannonical addresses will cause a protection fault



- Extending the PAE scheme with 64-bit PTE can map 48-bits of virtual memory ($9 \times 4 + 12 = 48$, 4-level mapping)

Representation of Virtual Address Space

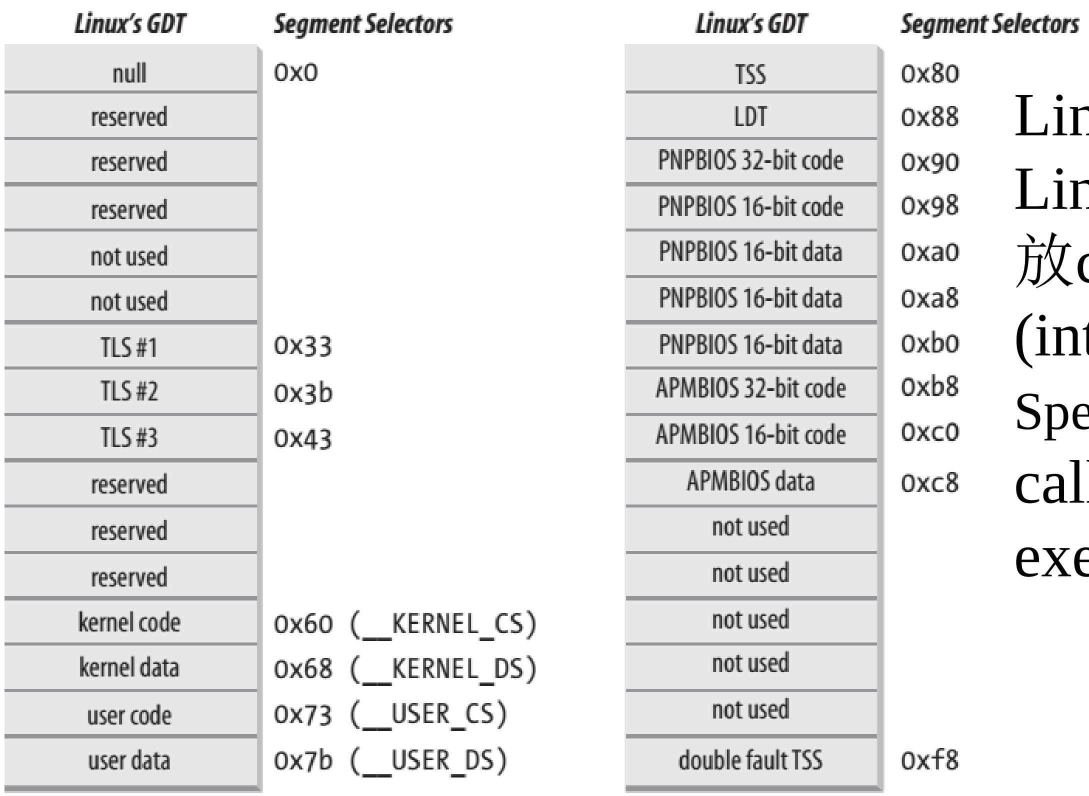


- Simplified Example
  - 16 page virtual address space
- Flags
  - P: Is entry in physical memory?
  - M: Has this part of VA space been mapped?

Linux存储管理概述

- **segmentation in Linux(主要4个段)**

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



Linux's GDT	Segment Selectors
null	0x0
reserved	
reserved	
reserved	
not used	
not used	
TLS #1	0x33
TLS #2	0x3b
TLS #3	0x43
reserved	
reserved	
reserved	
kernel code	0x60 (__KERNEL_CS)
kernel data	0x68 (__KERNEL_DS)
user code	0x73 (__USER_CS)
user data	0x7b (__USER_DS)

Linux's GDT	Segment Selectors
TSS	0x80
LDT	0x88
PNPBIOS 32-bit code	0x90
PNPBIOS 16-bit code	0x98
PNPBIOS 16-bit data	0xa0
PNPBIOS 16-bit data	0xa8
PNPBIOS 16-bit data	0xb0
APMBIOS 32-bit code	0xb8
APMBIOS 16-bit code	0xc0
APMBIOS data	0xc8
not used	
not used	
not used	
not used	
not used	
double fault TSS	0xf8

Linux的GDT有18项，其他预留; Linux的LDT只使用了2项，用于放call gate: a call gate for iBCS (intel Binary Compatibility Specification) executables, and a call gate for Solaris/x86 executables.

Linux存储管理概述

- **paging in Linux(4-level paging model)**
  - **For x86-64: 9+9+9+9+12=48bits**
  - **For IA32 without PAE: 10+0+0+10+12=32bits**

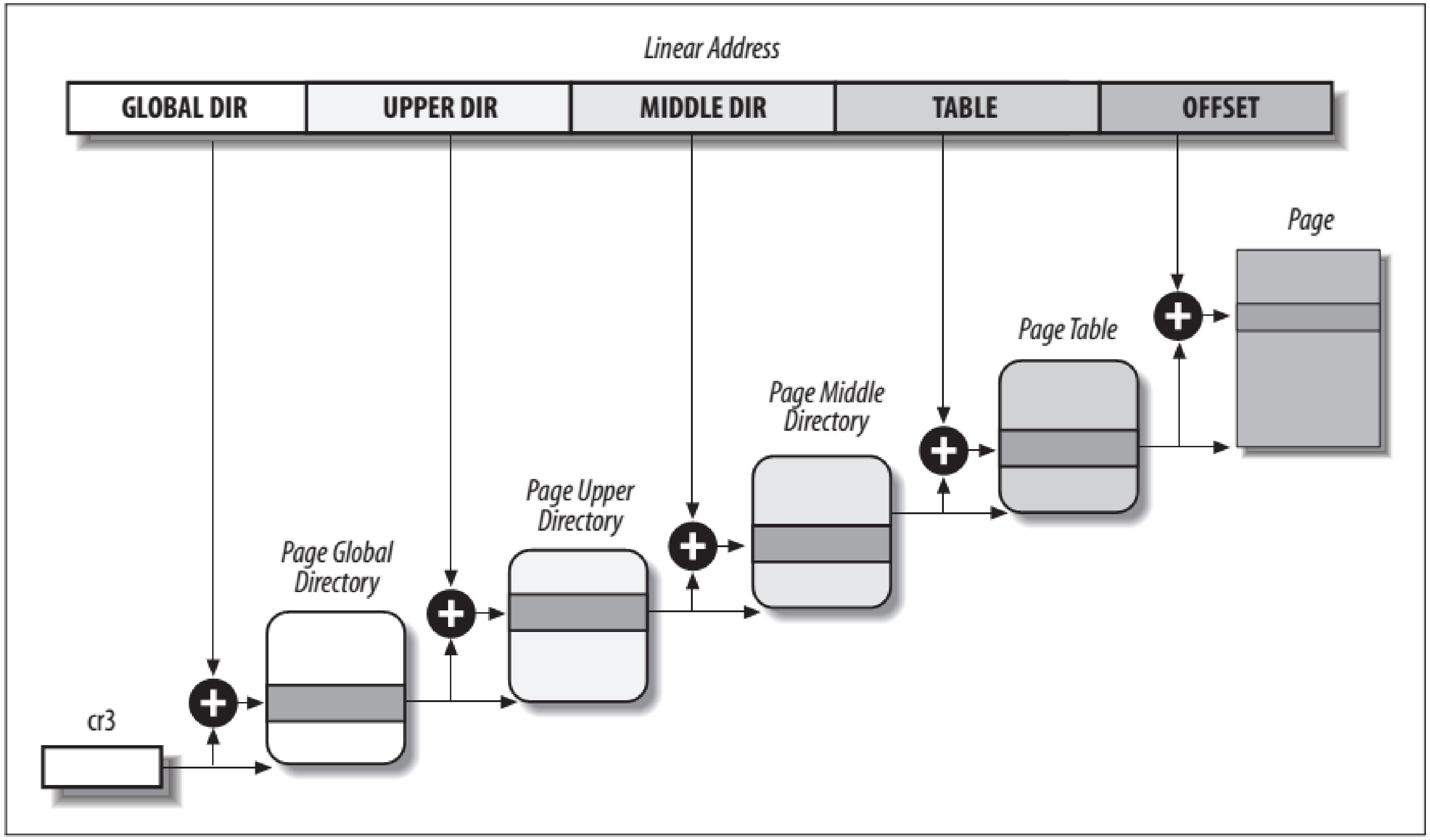


Figure 2-12. The Linux paging model

Linux物理内存管理

- 物理内存页面：页帧
  - 物理内存以页帧（**page frame**）为基本单位，**x86**页帧**4KB/4MB**，或**2MB**（**PAE**时）
- 区域：物理内存分为三个区域（**zone**）：
  - **ZONE_DMA**，低**16MB**，用于**DMA**
  - **ZONE_NORMAL**，**16MB~896MB**，直接被内核映射至**3GB+16MB**开始的虚地址
  - **ZONE_HIGHMEM**，高端内存，超过**896MB**以上部分，不能被内核直接映射
- 物理内存的管理
  - 基于**Buddy**算法的空闲物理内存页面的管理
  - 基于**slab**算法的内存对象管理

- 数据结构
  - ✓ 设置了一个mem_map[]数组管理内存页面page,
    - ✓ 其在系统初始化时由 free_area_init()函数创建。数组元素是一个个page结构体，每个page结构体对应一个物理页面。
    - ✓ page结构定义为mem_map_t类型，定义在/include/linux/mm.h中：

物理内存页面的管理

```
typedef struct page {
    struct page *next;
    struct page *prev;//把page结构体链接成一个双向循环链表
    struct inode *inode;
    unsigned long offset;
    struct page *prev_hash;
    struct page *next_hash;//把有关page结构体连成哈希表
    atomic_t count; //共享该页面的进程计数
    unsigned flags;
    unsigned dirty;//表示该页面是否被修改过
    unsigned age;//标志页面的“年龄”
    struct wait_queue  *wait;//等待该页资源的进程等待队列指针
    struct buffer_head * buffers;
    unsigned long swap_unlock_entry;
    unsigned long map_nr;//该页面page结构体在mem_map[]数组中的
下标值，也就是物理页面的页号
   } mem_map_t;
```

空闲物理内存页面的管理-Buddy算法

- Buddy算法
  - ➢ Linux对空闲物理内存页面管理采用Buddy算法。
  - – 把内存中所有页面按照2^n划分，其中n=0~10，每个内存空间按1个页面、2个页面、4个页面、8个页面、16个页面、32个页面…进行11次划分。
  - ✓ 划分后形成了大小不等的存储块，称为页面块，简称页块。包含1个页面的页块称为1页块，包含2个页面的称为2页块，依此类推。
  - ✓ 每种页块按前后顺序两两结合成一对Buddy“伙伴”
  - ✓ 系统按照Buddy关系把具有相同大小的空闲页面块组成页块组，即1页块组、2页块组……1024页块组。
  - ✓ 每个页块组用一个双向循环链表进行管理，共有11个链表，分别为1、2、4、、、1024页块链表。
- 分别挂到free_area[] 数组上。

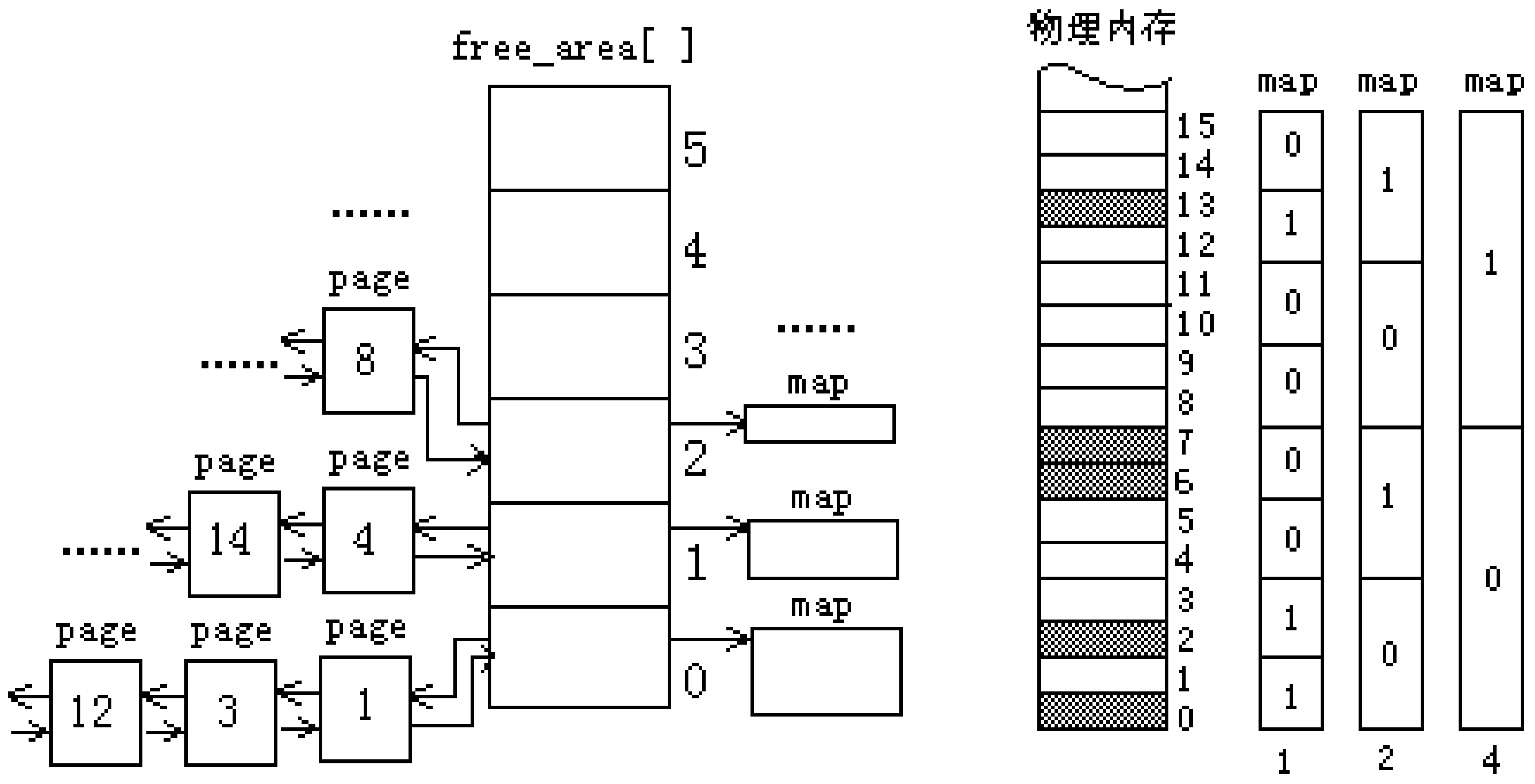
物理内存空间管理

- 位图数组
  - 标记内存页面使用情况，每一组每一位表示比邻的两个页面块的使用情况，依次类推。
    - 当一对Buddy的两个页面块中有一个是空闲的，而另一个全部或部分被占用时，该位置1。
    - 两个页面块都是空闲，或都被全部或部分占用时，对应位置0。

物理内存空间管理

- 内存分配和释放过程
  - ✓ 内存分配时，系统按照Buddy算法，根据请求的页面数在free_area[]对应的空闲页块组中搜索。
    - – 若请求页面数不是2的整数次幂，则按照稍大于请求数的2的整数次幂的值搜索相应的页面块组。
    - – 当相应页块组中没有可使用的空闲页面块时就查询更大一些的页块组，
    - – 在找到可用的空闲页面块后，分配所需页面。
    - – 当某一空闲页面块被分配后，若仍有剩余的空闲页面，则根据剩余页面的大小把它们加入到相应页块组中。
  - ✓ 内存页面释放时，系统将其作为空闲页面看待。
    - – 检查该页面的伙伴页（buddy）是否为空闲页块，若是，则合为一个连续的空闲区，并往上判断是否需要继续合并。

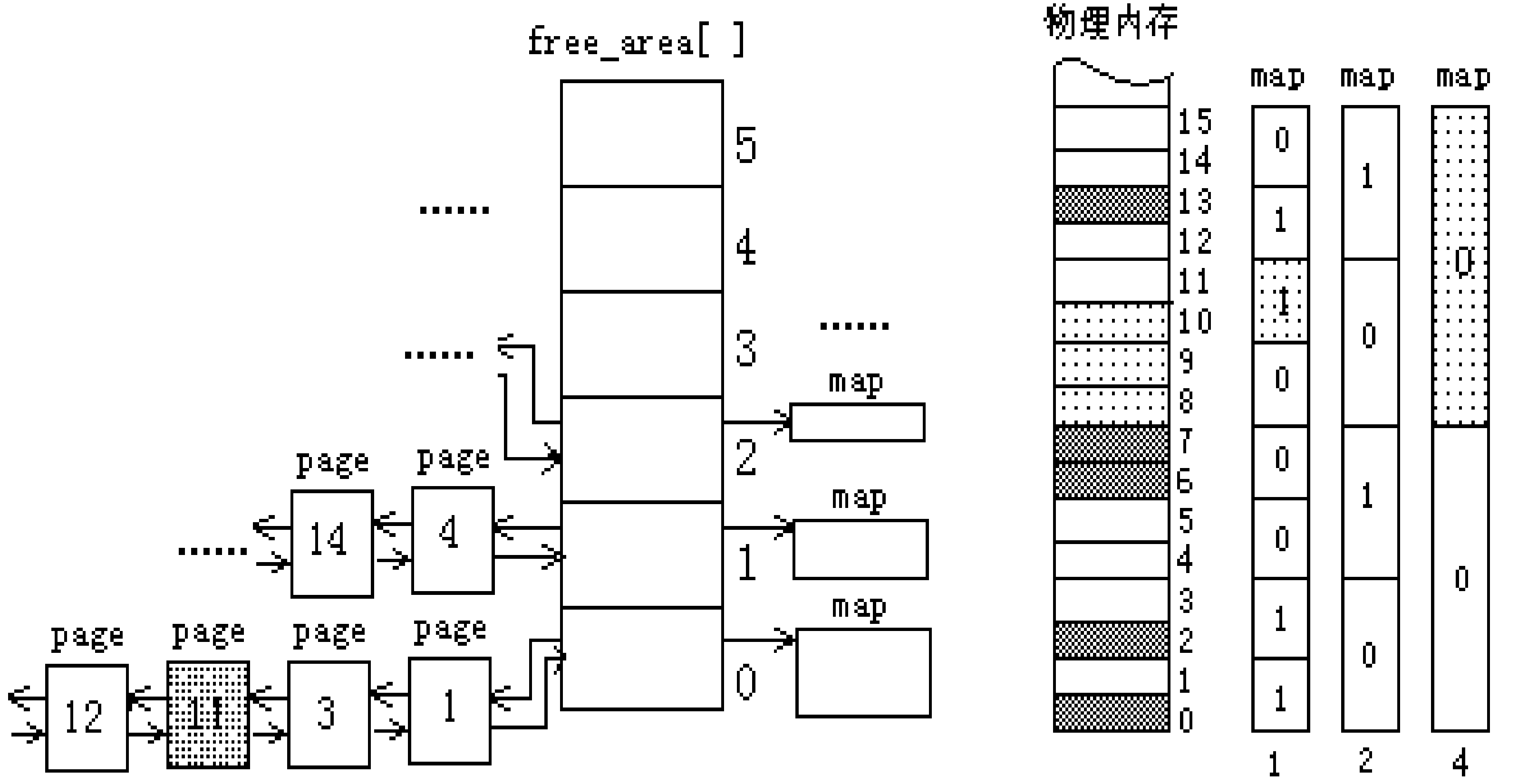
物理内存空间管理



内存分配的Buddy算法

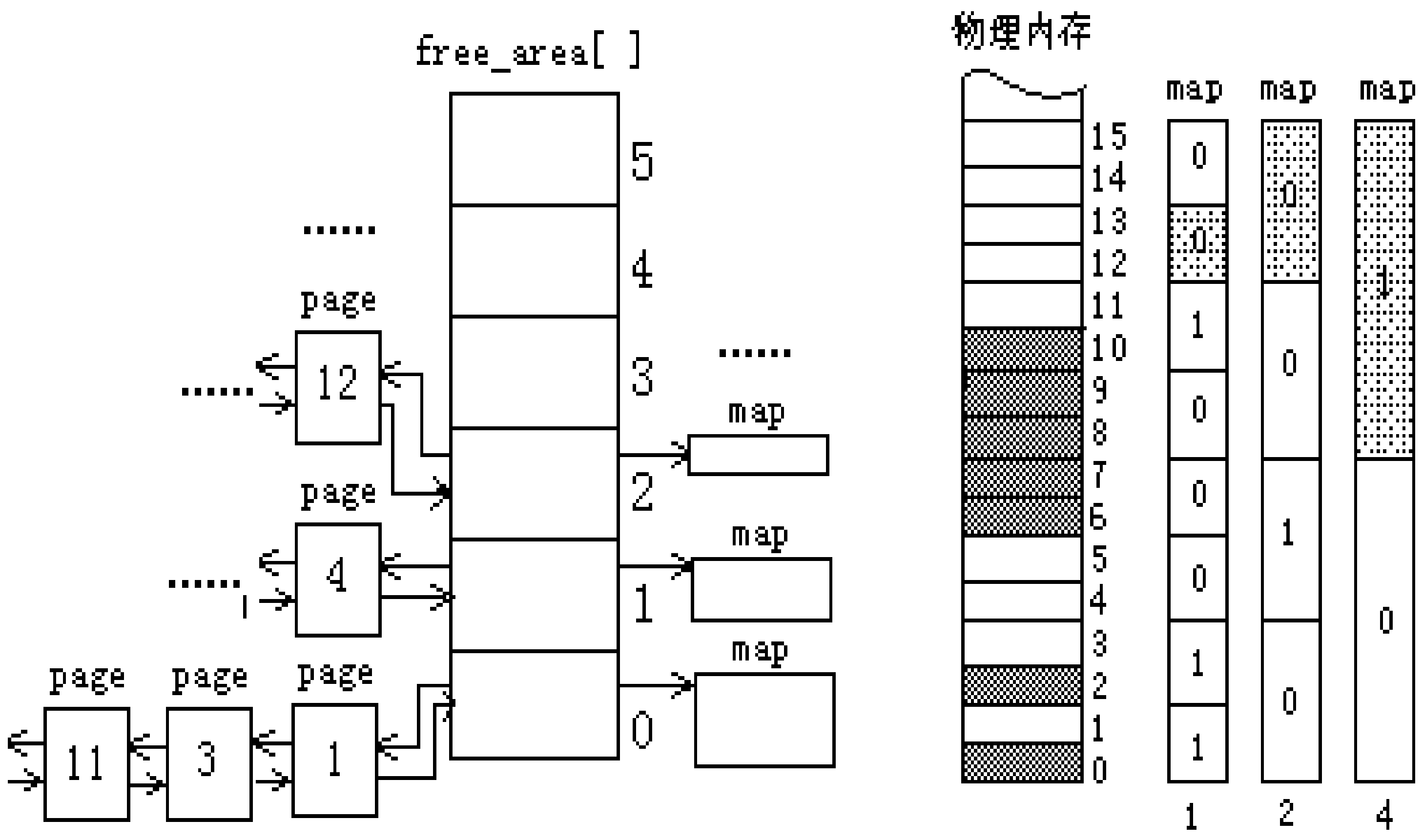
释放时看上一层的位图，如果为1，则表示伙伴是空闲的，就进行合并。

物理内存空间管理



8、9、10页面分配后的示意

物理内存空间管理



13号页面释放后

Internal Interfaces: Allocating Memory page(s)

- One mechanism for requesting pages: everything else on top of this mechanism:
  - Allocate contiguous group of pages of size 2^{order} bytes given the specified mask: (mm/page_alloc.c)

    ```
    struct page * alloc_pages(gfp_t gfp_mask,
                              unsigned int order)
    ```

  - Allocate one page:

    ```
    struct page * alloc_page(gfp_t gfp_mask)
    ```

  - Convert page to logical address (assuming mapped):

    ```
    void * page_address(struct page *page)
    ```
- Also routines for freeing pages
- Zone allocator uses “buddy” allocator that tries to keep memory un-fragmented
- Allocation routines pick from proper zone, given flags

Allocation flags

- Possible allocation type flags:

– GFP_ATOMIC:	Allocation high-priority and must never sleep. Use in interrupt handlers, top halves, while holding locks, or other times cannot sleep
– GFP_NOWAIT:	Like GFP_ATOMIC, except call will not fall back on emergency memory pools. Increases likely hood of failure
– GFP_NOIO:	Allocation can block but must not initiate disk I/O.
– GFP_NOFS:	Can block, and can initiate disk I/O, but will not initiate filesystem ops.
– GFP_KERNEL:	Normal allocation, might block. Use in process context when safe to sleep. This should be default choice
– GFP_USER:	Normal allocation for processes
– GFP_HIGHMEM:	Allocation from ZONE_HIGHMEM
– GFP_DMA	Allocation from ZONE_DMA. Use in combination with a previous flag

Page Frame Reclaiming Algorithm (PFRA)

- Several entry points:
  - Low on Memory Reclaiming: The kernel detects a “low on memory” condition
  - Hibernation reclaiming: The kernel must free memory because it is entering in the suspend-to-disk state
  - Periodic reclaiming: A kernel thread is activated periodically to perform memory reclaiming, if necessary
- Low on Memory reclaiming:
  - Start flushing out dirty pages to disk
  - Start looping over all memory nodes in the system
    - ✓ try_to_free_pages()
    - ✓ shrink_slab()
    - ✓ pdflush kenel thread writing out dirty pages
- Periodic reclaiming:
  - Kswapd kernel thread: checks if number of free page frames in some zone has fallen below pages_high watermark
  - Each zone keeps two LRU lists: Active and Inactive
    - ✓ Each page has a last-chance algorithm with 2 count
    - ✓ Active page lists moved to inactive list when they have been idle for two cycles through the list
    - ✓ Pages reclaimed from Inactive list

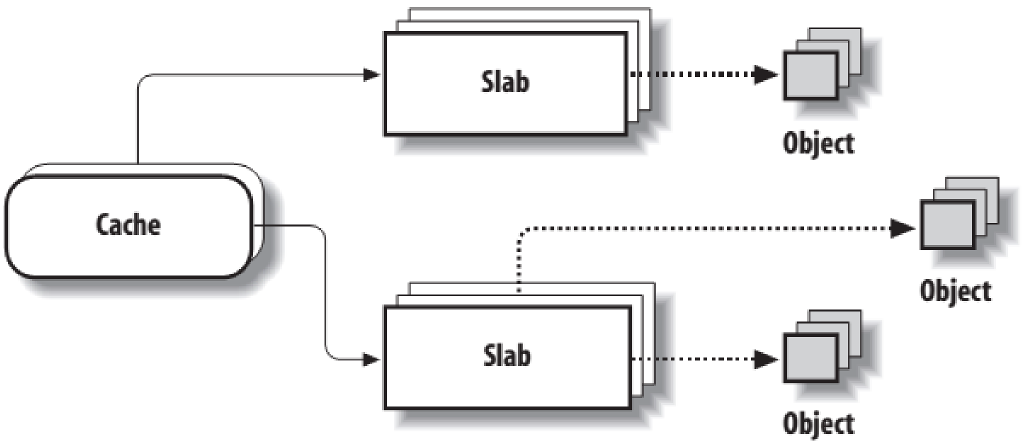
物理内存空间管理-Slab分配器

- Buddy算法
  - 采用buddy算法，解决了外碎片问题，这种方法适合大块内存请求，不适合小内存区请求。如：几十个或者几百个字节。
  - Linux2.0采用传统内存分区算法，按几何分布提供内存区大小，内存区以2的幂次方为单位。虽然减少了内碎片，但没有显著提高系统效率。
- Slab**算法**
  - 对象内存的申请和释放
  - 充分利用了空间，减少了内部碎片
  - 管理局部化，尽可能减少了与buddy分配器打交道，提高了效率。

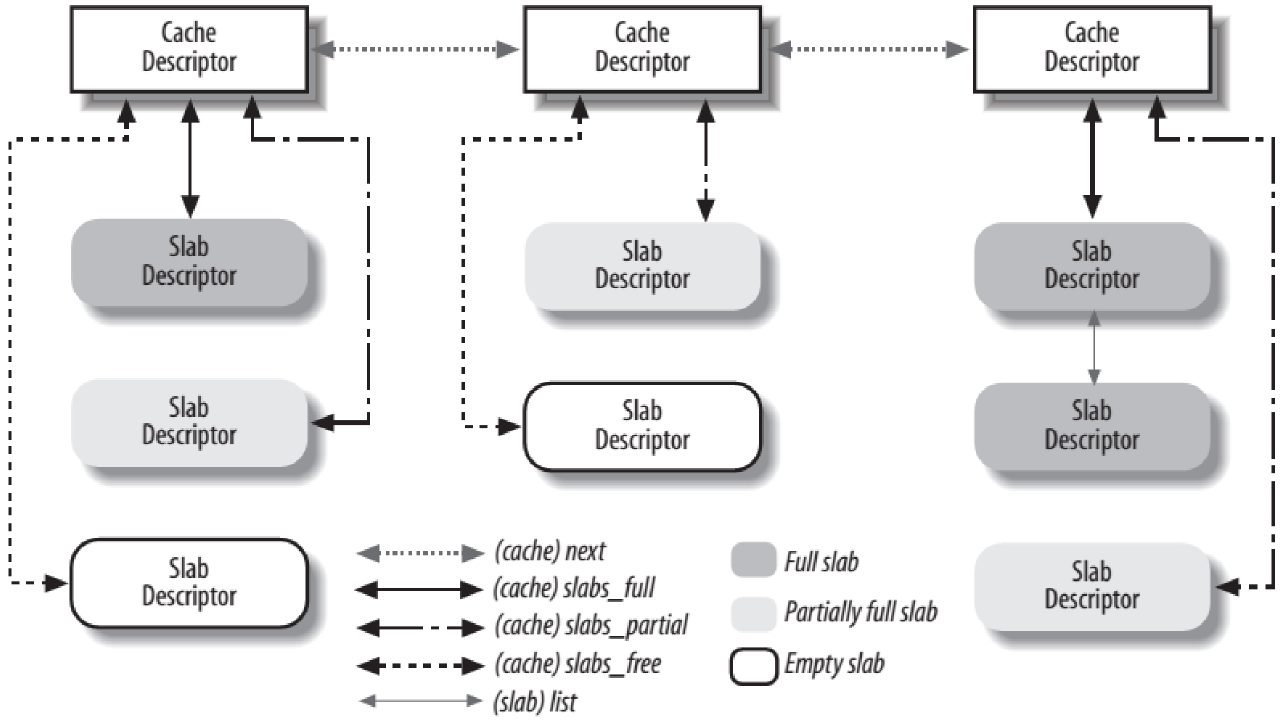
物理内存空间管理

- Linux 2.4开始采用了slab分配器算法(2.6.22 以后引入了SLUB)
  - 该算法比传统的分配器算法有更好性能和内存利用率，最早在solaris2.4上使用。
- Slab分配器思想
  - 对象的申请和释放通过slab分配器来管理。
  - slab分配器有一组高速缓存（cache），每个高速缓存保存同一种类型的对象，如i-node、PCB等。（还有一些通用对象，如32B，64B，…，128KB）
  - 内核从这些高速缓存中分配和释放对象。
  - 每种对象的高速缓存由一连串slab构成，每个slab由一个或者多个连续的物理页面组成。这些页面中包含了已分配的缓存对象，也包含了空闲对象。

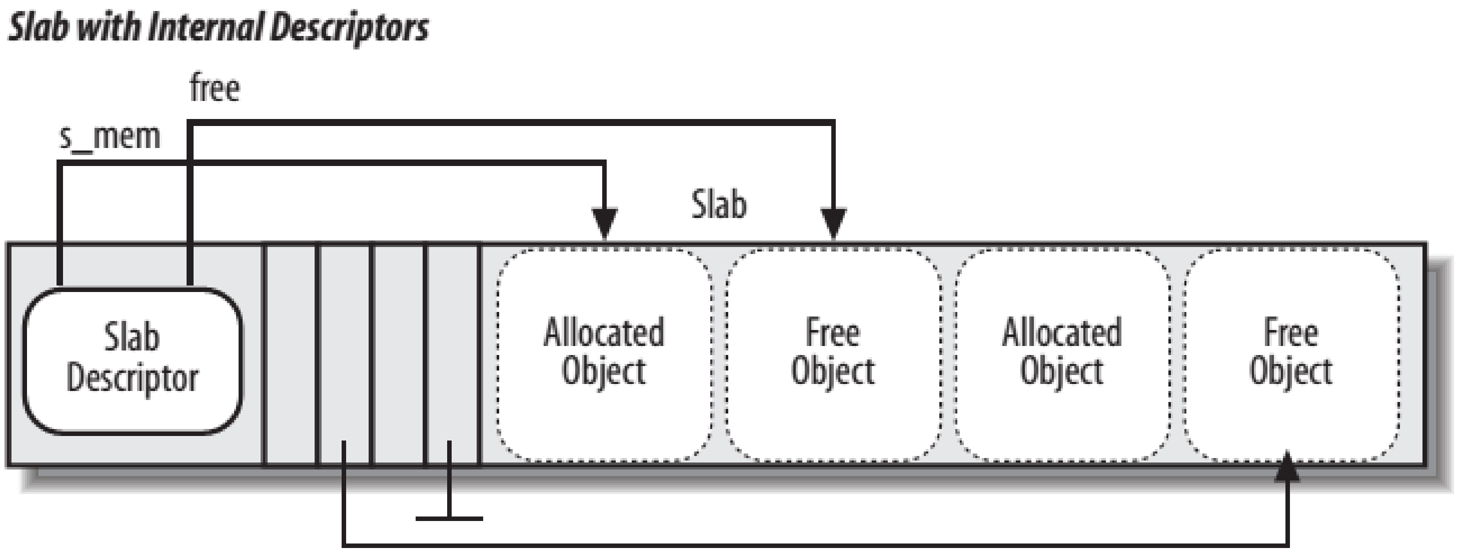
cache与slab

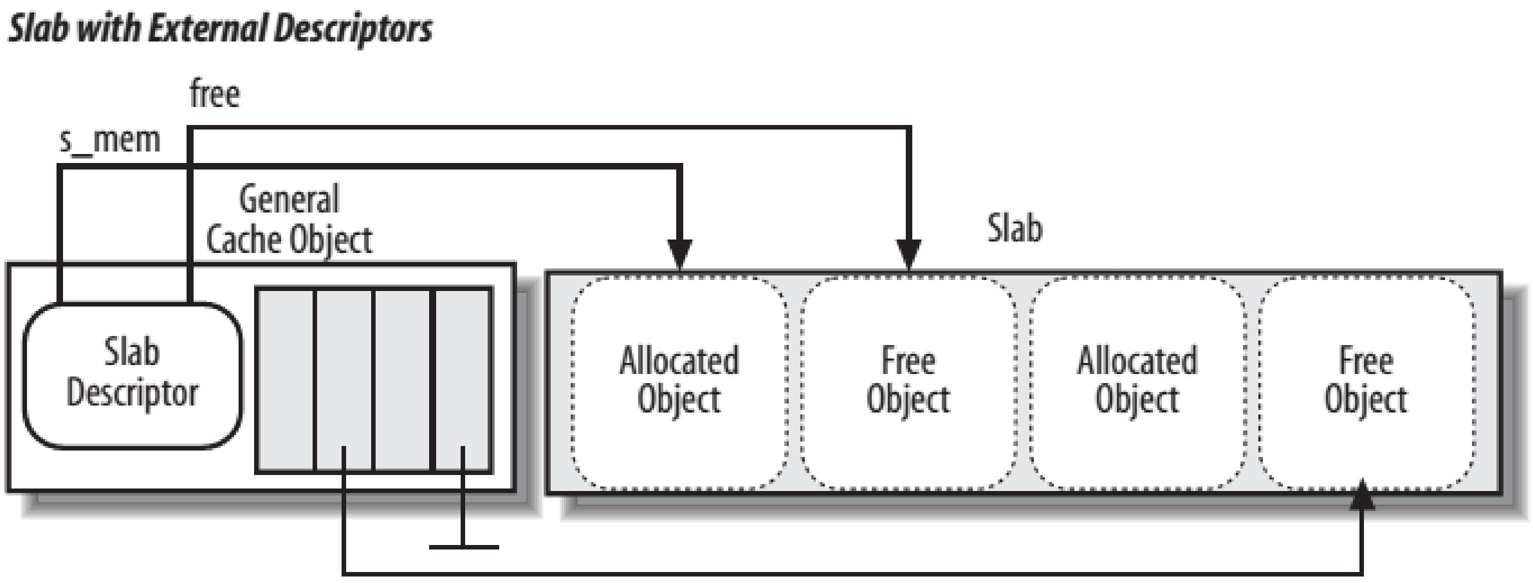


- 数据结构
  - 高速缓存
  - Struct kmem_cache_s
  - 高速缓存中的每个slab数据结构
  - Struct kmem_slab_s



Slab descriptor

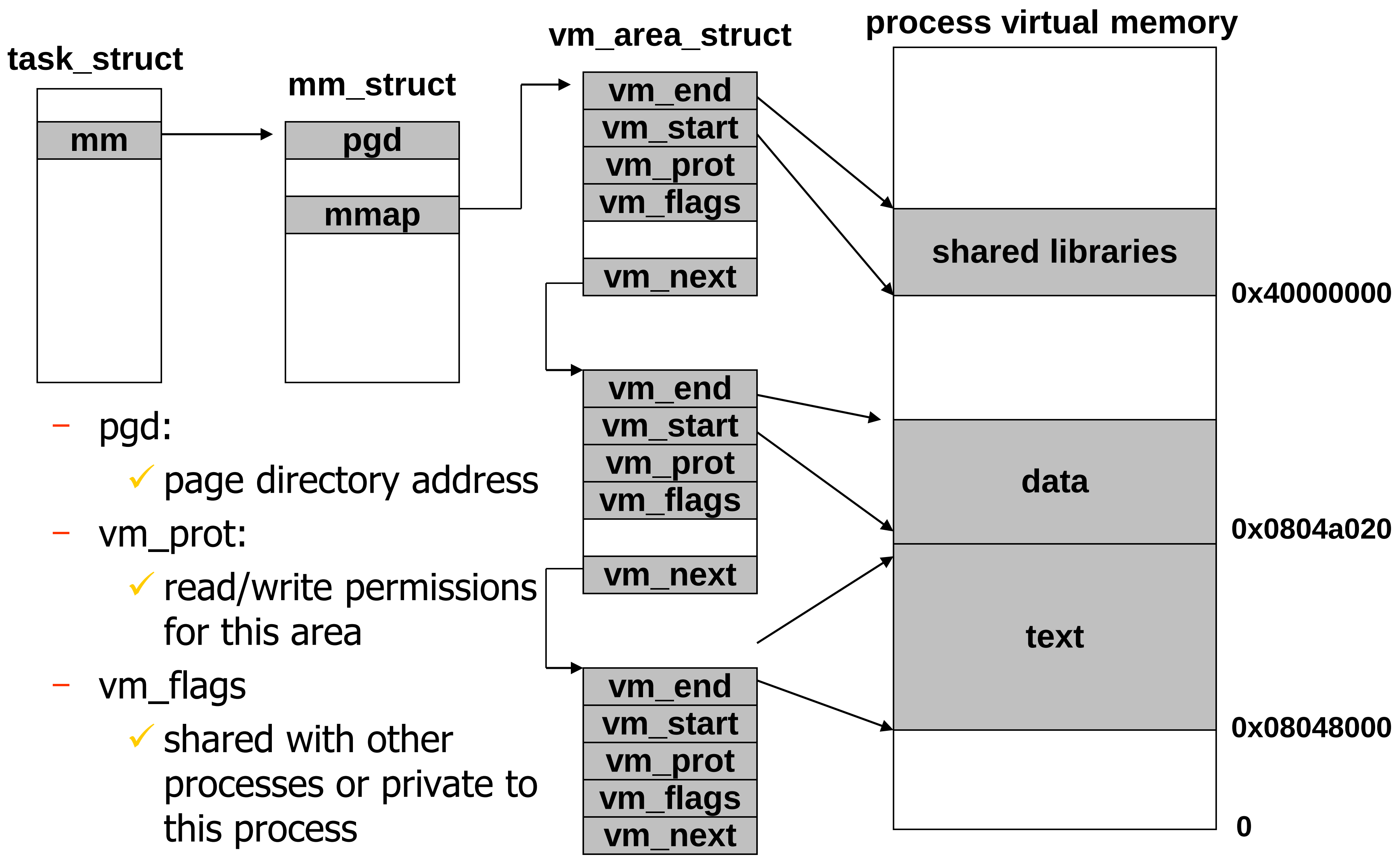




物理内存空间管理

- 创建slab过程
- Slab分配器调用Kmem_cache_grow() 函数为缓存分配一个新的slab。
  - 调用kmem_getpages()获取一组连续内存页框；
  - 调用kmem_cache_slabmgamt()分配一个新的slab数据结构；
  - 调用kmem_cache_init_objs（）为新slab中包含的所有对象定义构造方法；
  - 调用kmem_slab_link_end()将新的slab插入到这个高速缓存的双向链表的末尾。

Linux虚存管理: VMA-Virtual Memory Area

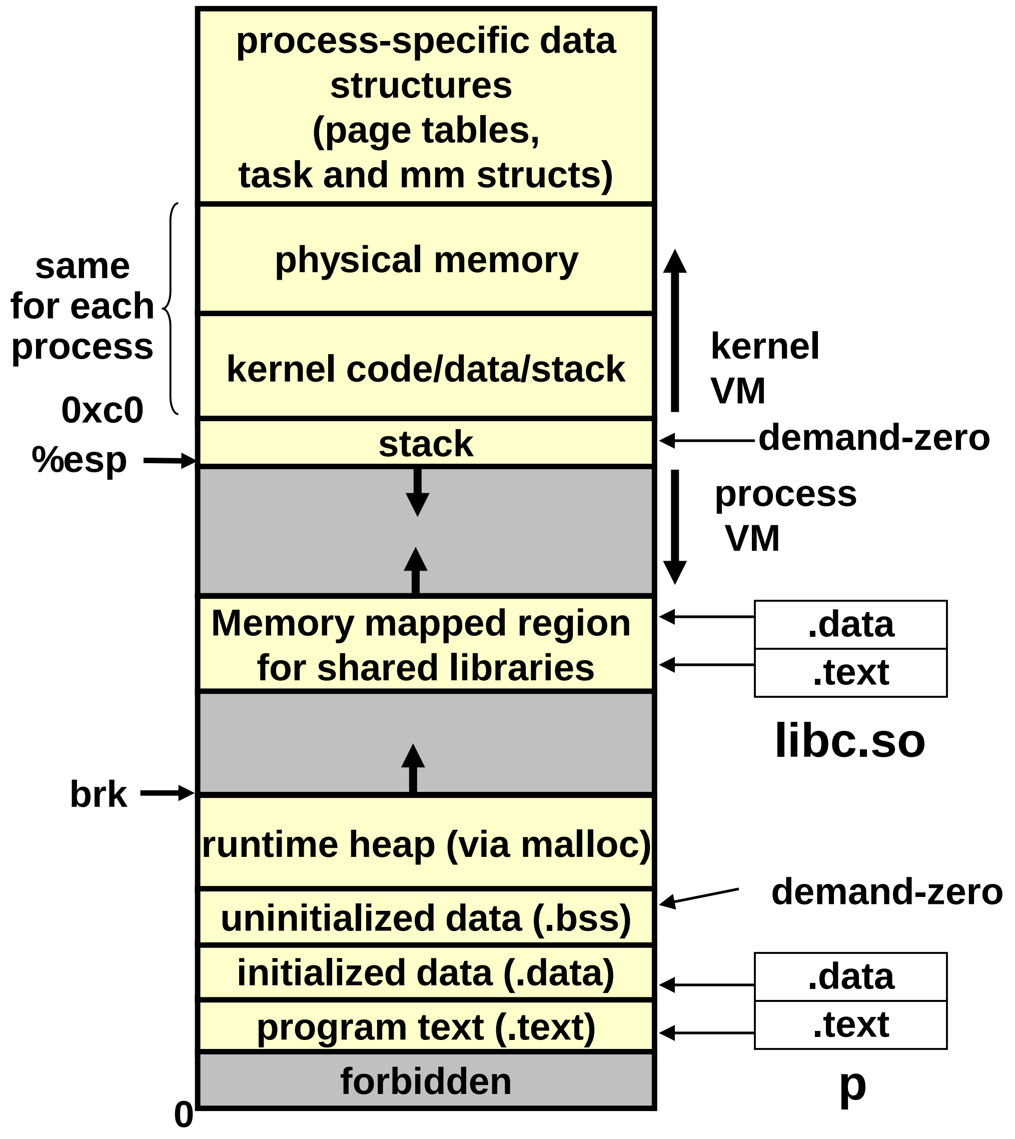


- pgd:
  - ✓ page directory address
- vm_prot:
  - ✓ read/write permissions for this area
- vm_flags
  - ✓ shared with other processes or private to this process

Fork() Revisited

- To create a new process using `fork()`:
  - make copies of the old process's mm_struct, vm_area_structs, and page tables.
    - ✓ at this point the two processes are sharing all of their pages.
    - ✓ How to get separate spaces without copying all the virtual pages from one space to another?
      - "copy on write" technique.
  - copy-on-write
    - ✓ make pages of writeable areas read-only
    - ✓ flag vm_area_structs for these areas as private "copy-on-write".
    - ✓ writes by either process to these pages will cause page faults.
      - fault handler recognizes copy-on-write, makes a copy of the page, and restores write permissions.
  - result:
    - ✓ copies are deferred until absolutely necessary (i.e., when one of the processes tries to modify a shared page).

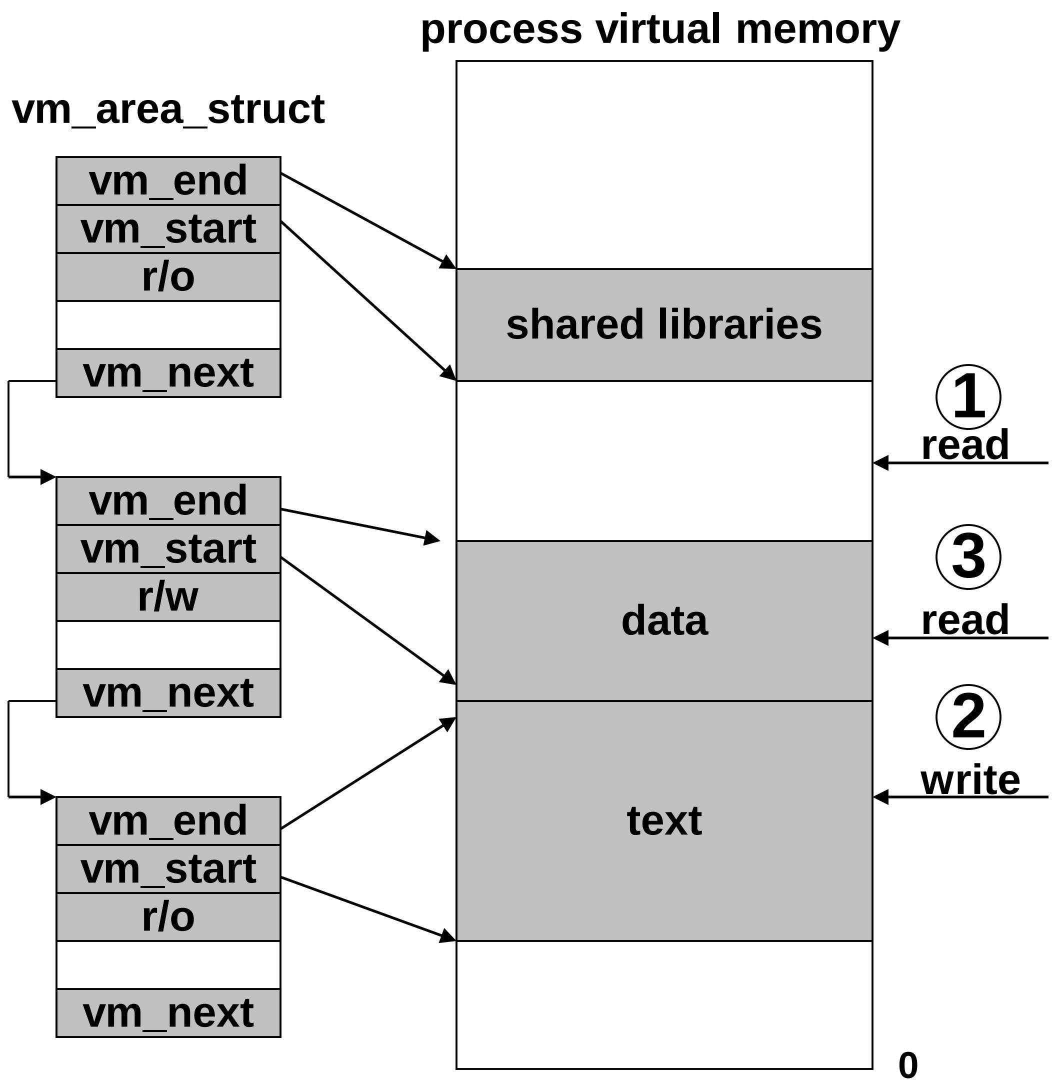
Exec() Revisited



- To run a new program p in the current process using `exec()`:
  - free vm_area_structs and page tables for old areas.
  - create new vm_area_structs and page tables for new areas.
    - ✓ stack, bss, data, text, shared libs.
    - ✓ text and data backed by ELF executable object file.
    - ✓ bss and stack initialized to zero.
  - set PC to entry point in .text
    - ✓ Linux will swap in code and data pages as needed.

bss: block started by symbol,未初始化的全局变量

Linux Page Fault Handling



- Is the VA legal?
  - i.e. is it in an area defined by a vm_area_struct?
  - if not then signal segmentation violation (e.g. (1))
- Is the operation legal?
  - i.e., can the process read/write this area?
  - if not then signal protection violation (e.g., (2))
- If OK, handle fault
  - e.g., (3)

Memory Mapping

- Creation of new VM *area* done via "memory mapping"
  - create new vm_area_struct and page tables for area
  - area can be backed by (i.e., get its initial values from) :
    - ✓ regular file on disk (e.g., an executable object file)
      - initial page bytes come from a section of a file
    - ✓ nothing (e.g., bss)
      - initial page bytes are zeros
  - dirty pages are swapped back and forth between a special swap file.
- Key point: no virtual pages are copied into physical memory until they are referenced!
  - known as "demand paging"
  - crucial for time and space efficiency

User-Level Memory Mapping

- `void *mmap(void *start, int len,`
  `int prot, int flags, int fd, int offset)`
  - map `len` bytes starting at offset `offset` of the file specified by file description `fd`, preferably at address `start` (usually 0 for don't care).
    - ✓ `prot`: MAP_READ, MAP_WRITE
    - ✓ `flags`: MAP_PRIVATE, MAP_SHARED
  - return a pointer to the mapped area.
  - Example: fast file copy
    - ✓ useful for applications like Web servers that need to quickly copy files.
    - ✓ `mmap` allows file transfers without copying into user space.

mmap() Example: Fast File Copy

```
#include <unistd.h>
#include <sys/mman.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

/*
 * mmap.c - a program that uses
  mmap to copy itself to stdout
 */
```

```
int main() {
  struct stat stat;
  int i, fd, size;
  char *bufp;

  /* open the file & get its size*/
  fd = open("./mmap.c", O_RDONLY);
  fstat(fd, &stat);
  size = stat.st_size;
 /* map the file to a new VM area*/
  bufp = mmap(0, size, PROT_READ,
    MAP_PRIVATE, fd, 0);

  /* write the VM area to stdout */
  write(1, bufp, size);
}
```

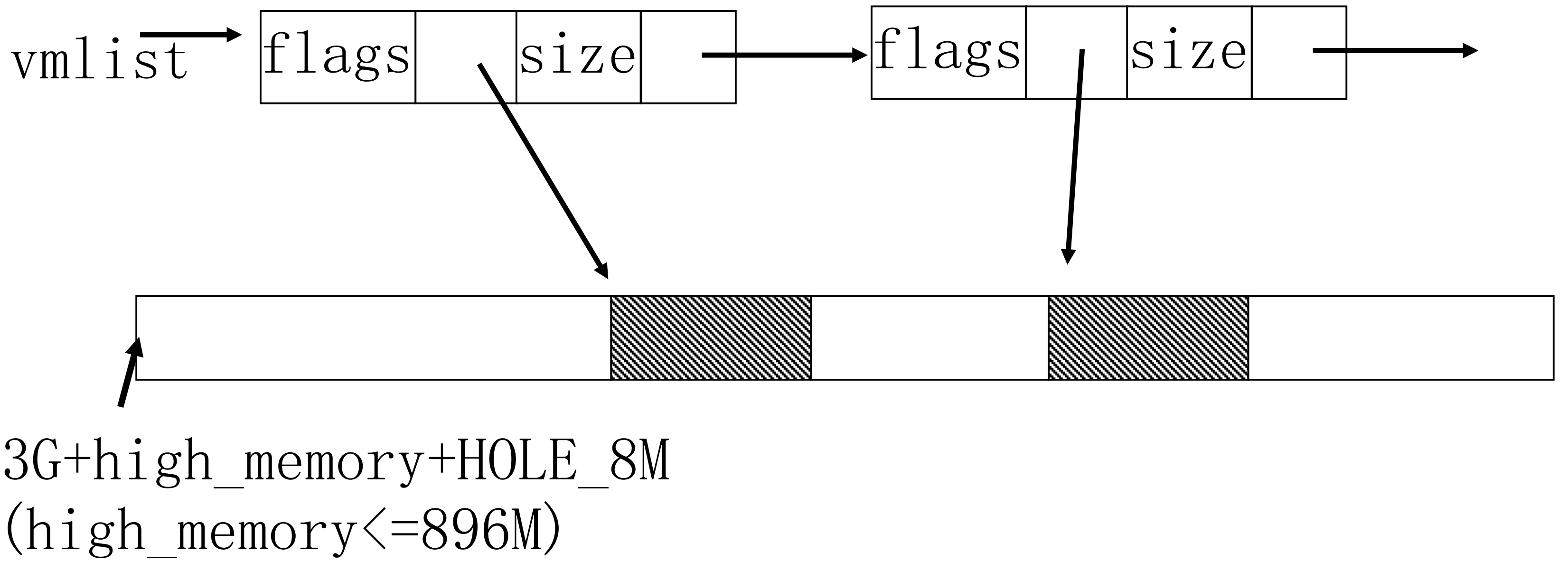
内核态虚存的申请与释放

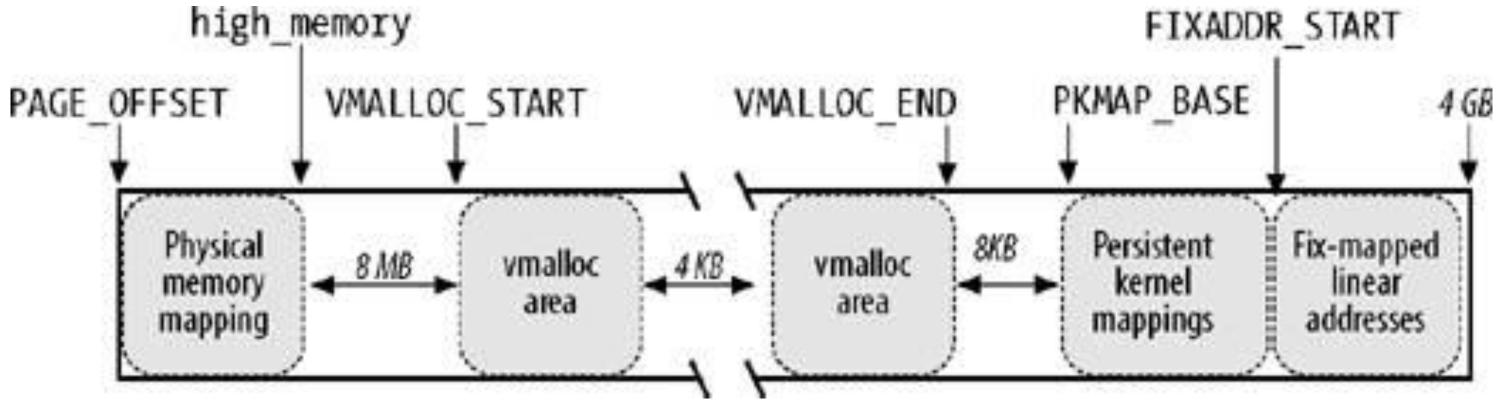
- 内核态有时需要申请大片的内存，如用于交换、内核模块、I/O缓冲等，则采用申请虚地址空间连续、物理页面不连续的方法
- 可分配的虚拟空间在3G+high_memory+HOLE_8M以上高端，由vmlist链表管理
- 申请与释放(mm/vmalloc.c)

  vmalloc()和vfree()
- vmlist链表的节点类型(include/linux/vmalloc.h)

```
struct vm_struct {
        unsigned long flags; /* 虚拟内存块的占用标志 */
        void * addr;         /* 虚拟内存块的起始地址 */
        unsigned long size; /* 虚拟内存块的长度 */
        struct vm_struct * next; /* 下一个虚拟内存块 */
};
static struct vm_struct * vmlist = NULL;
```

3G+high_memory+HOLE_8M以上高端空间





The end!