中国人民大学 信息学院
RENMIN UNIVERSITY OF CHINA SCHOOL OF INFORMATION

操作系统内核分析与安全

4. 内存管理与进程地址空间

授课教师：游伟 副教授

授课时间：周五14:00 – 16:30（立德楼807）

课程主页：https://www.youwei.site/course/kernel

目录

4.1.1 从硬件角度看内存管理

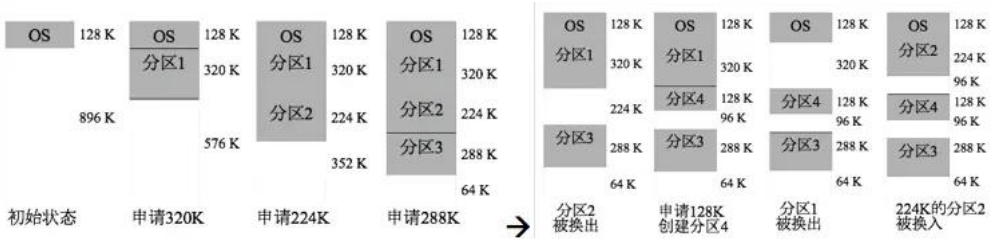
- 内存管理的“远古时代”
  - 单道程序的内存管理：整个系统只有一个用户进程和一个操作系统，用户程序独占整个用户空间，实现简单，存储器利用率极低
  - 多道程序的内存管理：系统有多个进程并发执行，内存管理出现了固定分区和动态分区两种技术

固定分区

- 由操作系统或系统管理员预先将内存划分成若干个分区。在系统运行过程中，分区边界不再改变。
- 分配时，找一个空闲且足够大的分区。如果没有合适的分区：①让申请者等待；②先换出某分区的内容，再将其分配出去。
- 缺点：内存利用率低，产生内部碎片；尺寸和分区数量难以确定。

动态分区

- 初始情况下，把所有的空闲内存看成一个大分区。
- 分配时，按申请的尺寸，找一块足够大的空闲内存分区，临时从中划出一块构成新分区
- 回收时，尽可能与邻近的空闲分区合并。在内存紧缺时，可将某个选定的分区换出。
- 缺点：不会产生内部碎片，但会产生外部碎片

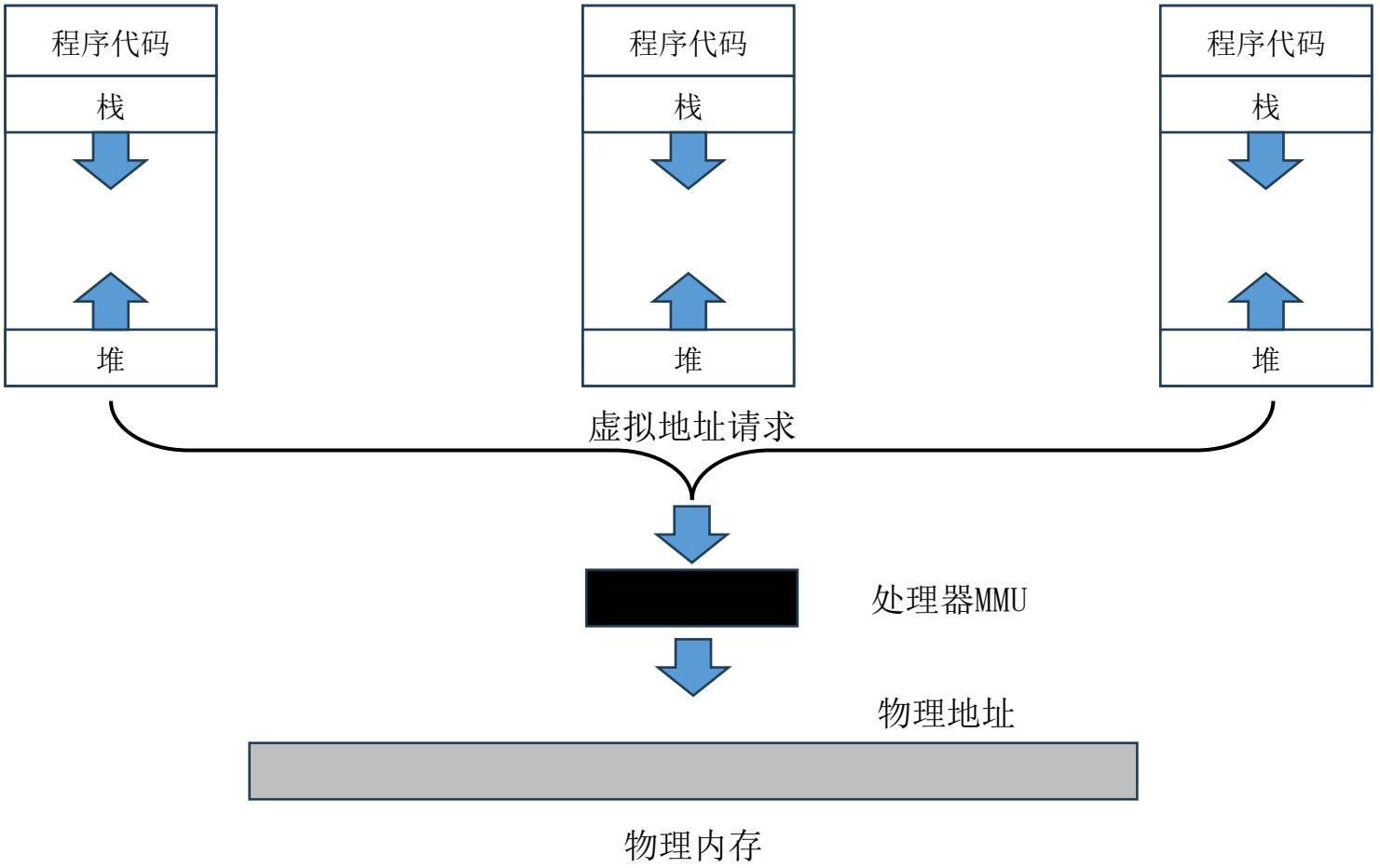


远古内存管理的问题

- 进程地址空间保护问题：所有用户进程都可访问全部物理内存，恶意程序或普通程序的错误操作，可以修改其它程序的内存数据
- 内存使用效率低：如果即将运行的进程所需要的内存空间不足，就需要选择一个进程整体换出，导致大量的数据需要换入换出
- 程序运行地址重定位问题：进程直接使用物理地址访问内存，进程在每次换入换出时运行的地址是不固定的，内存数据访问和指令跳转语句需要使用重定位技术

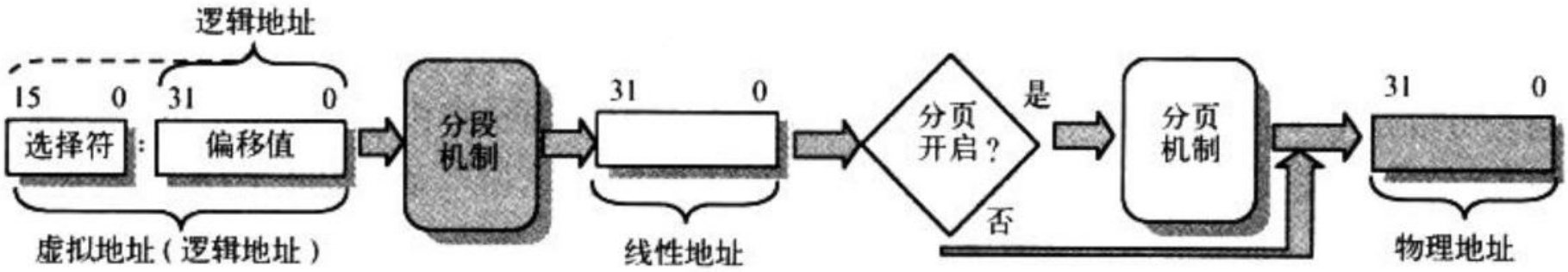
地址空间的抽象：虚拟地址

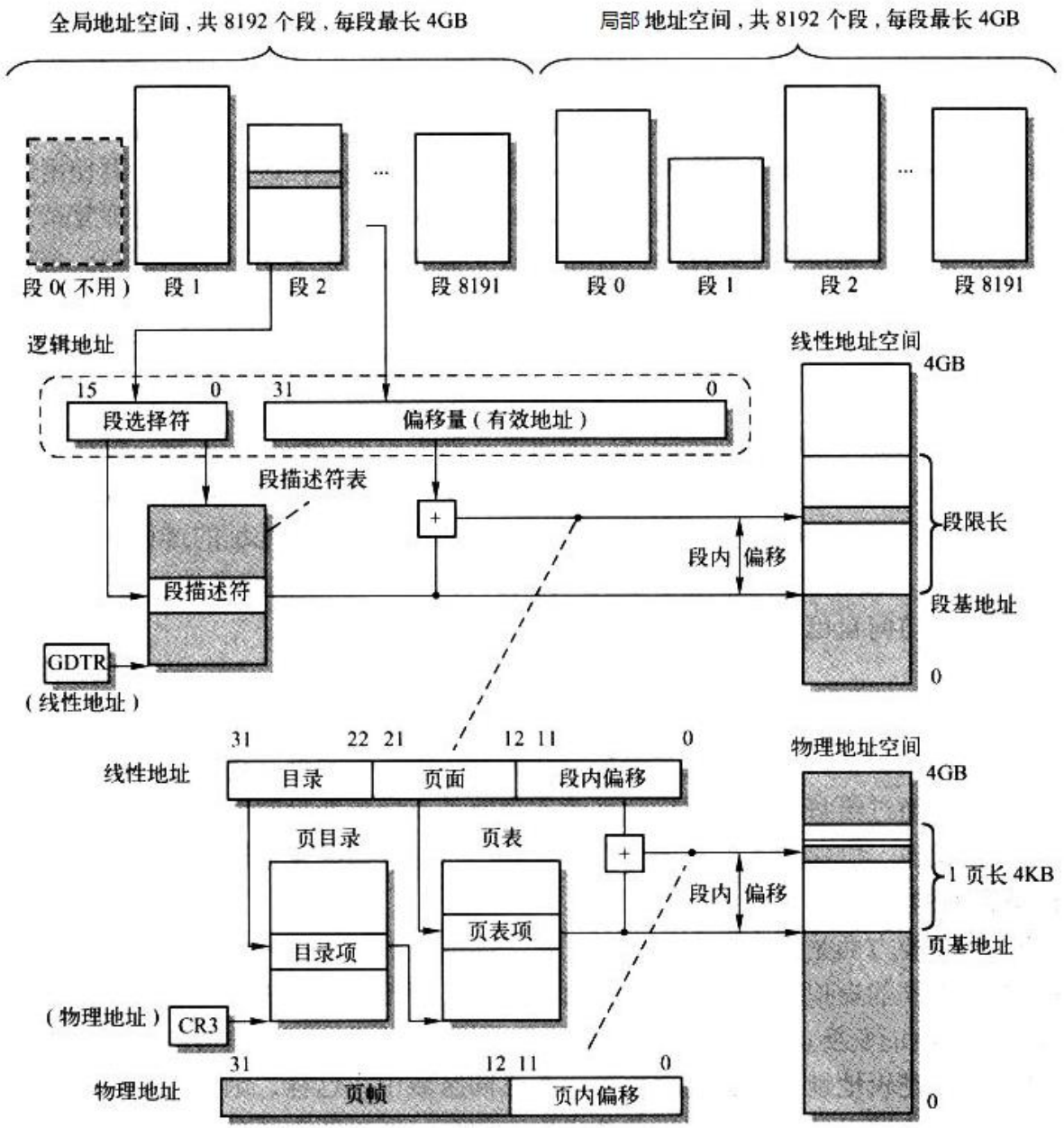
- 每个进程感觉自己拥有了整个地址空间



4.1.1 从硬件角度看内存管理

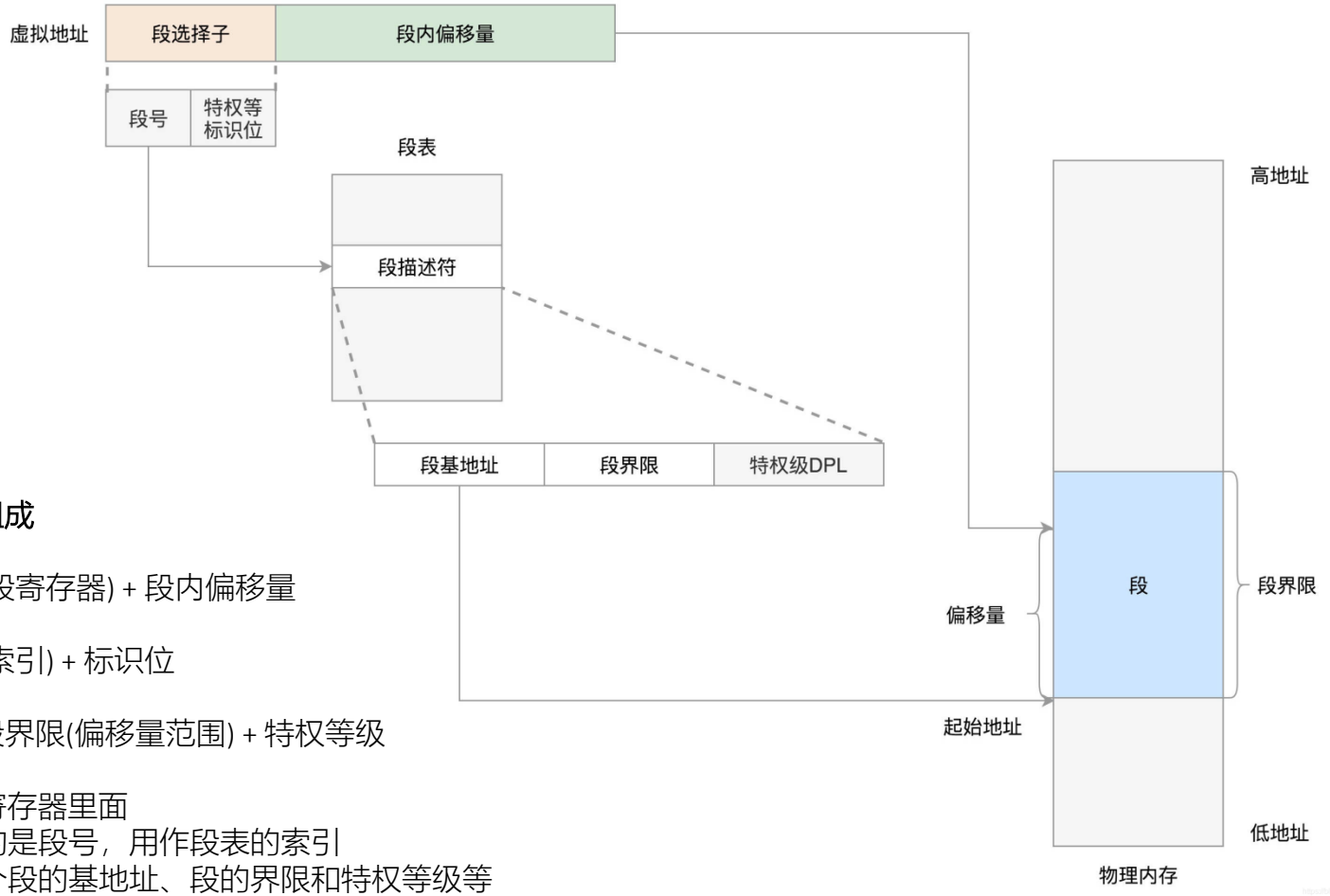
- i386体系结构的内存管理
  - 逻辑地址：包含在机器语言指令中用来指定一个操作数或一条指令的地址
  - 线性地址：是逻辑地址到物理地址变换之间的中间层
  - 物理地址：物理内存真正的地址，相当于内存中每个存储单元的门牌号





逻辑地址、线性地址与物理地址之间的变换

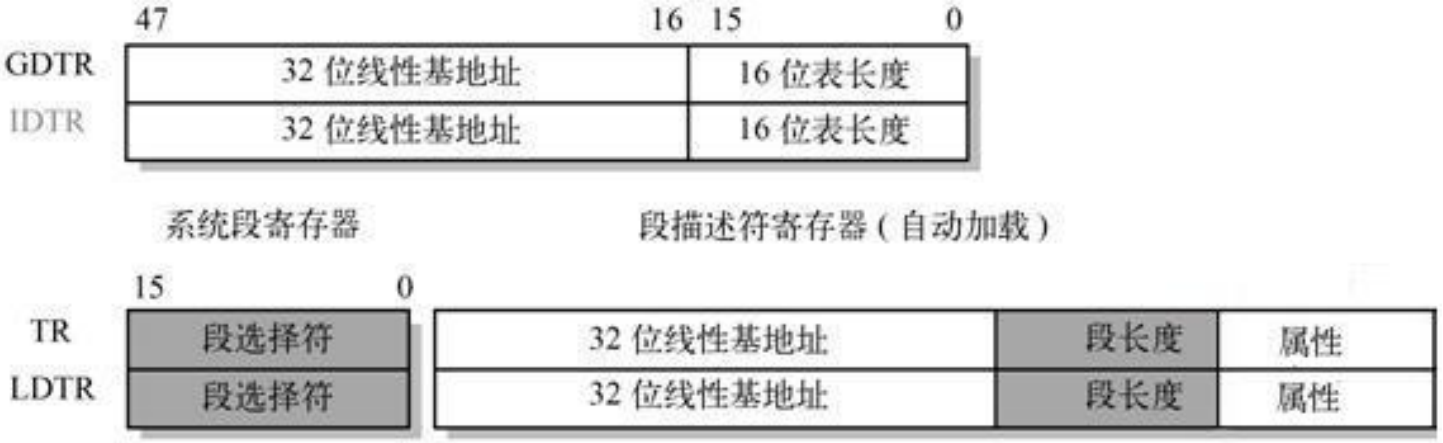
分段机制



分段机制下虚拟地址的组成

- 虚拟地址 = 段选择子(段寄存器) + 段内偏移量
- 段选择子 = 段号(段表索引) + 标识位
- 段表 = 物理基地址 + 段界限(偏移量范围) + 特权等级
- 段选择子：保存在段寄存器里面
  段选择子里面最重要的是段号，用作段表的索引
  段表里面保存的是这个段的基地址、段的界限和特权等级等
- 段内偏移量
  虚拟地址中的段内偏移量应该位于 0 和段界限之间
  段内偏移量是合法的，就将段基地址加上段内偏移量得到物理内存地址

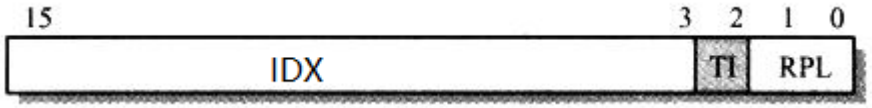
内存管理寄存器



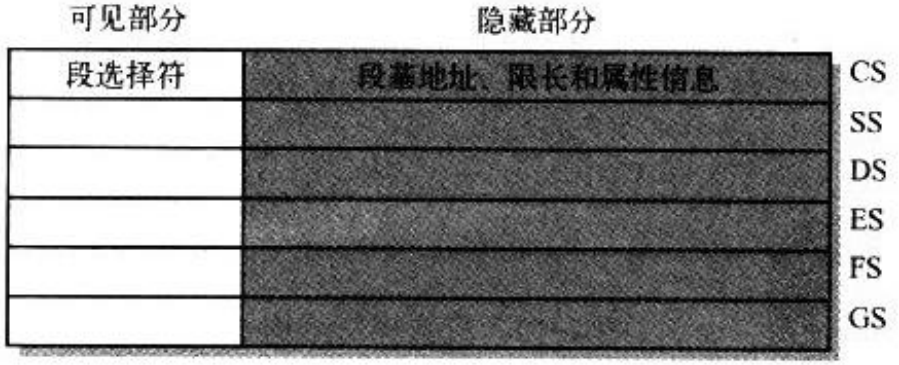
- GDTR中存放全局描述符表的线性地址和长度
- IDTR中存放中断描述符表的线性地址和长度
- LDTR中存放一个段选择器，该段选择器指向全局描述符表中LDT段描述符所在的表项
- TR中存放一个段选择器，该段选择器指向全局描述符表中TSS段描述符所在的表项

段选择符与段寄存器

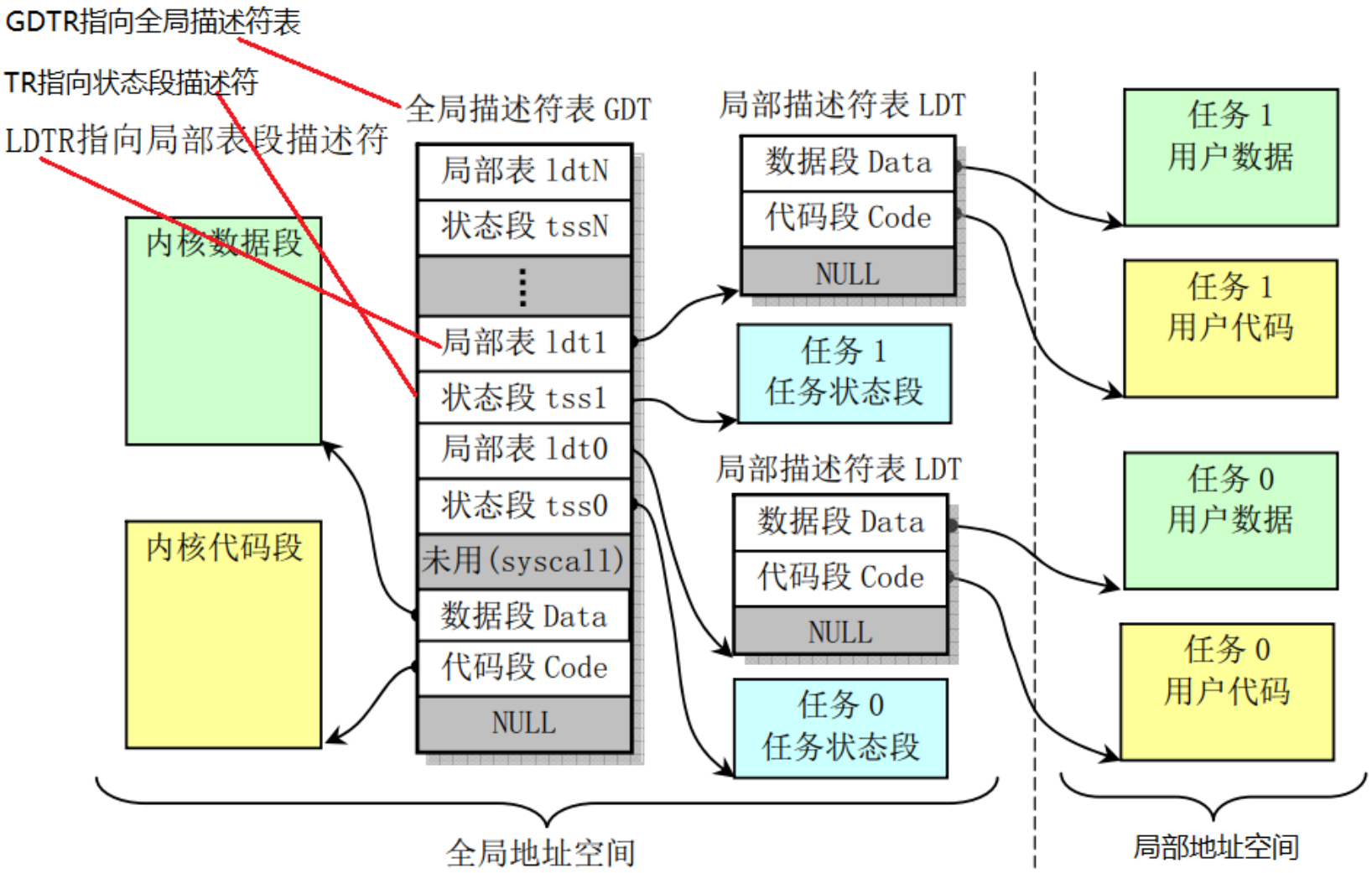
- 段选择符
  - 请求特权级RPL（Requested Privilege Level）
  - 表指示标志TI（Table Index）
  - 索引值IDX（Index）



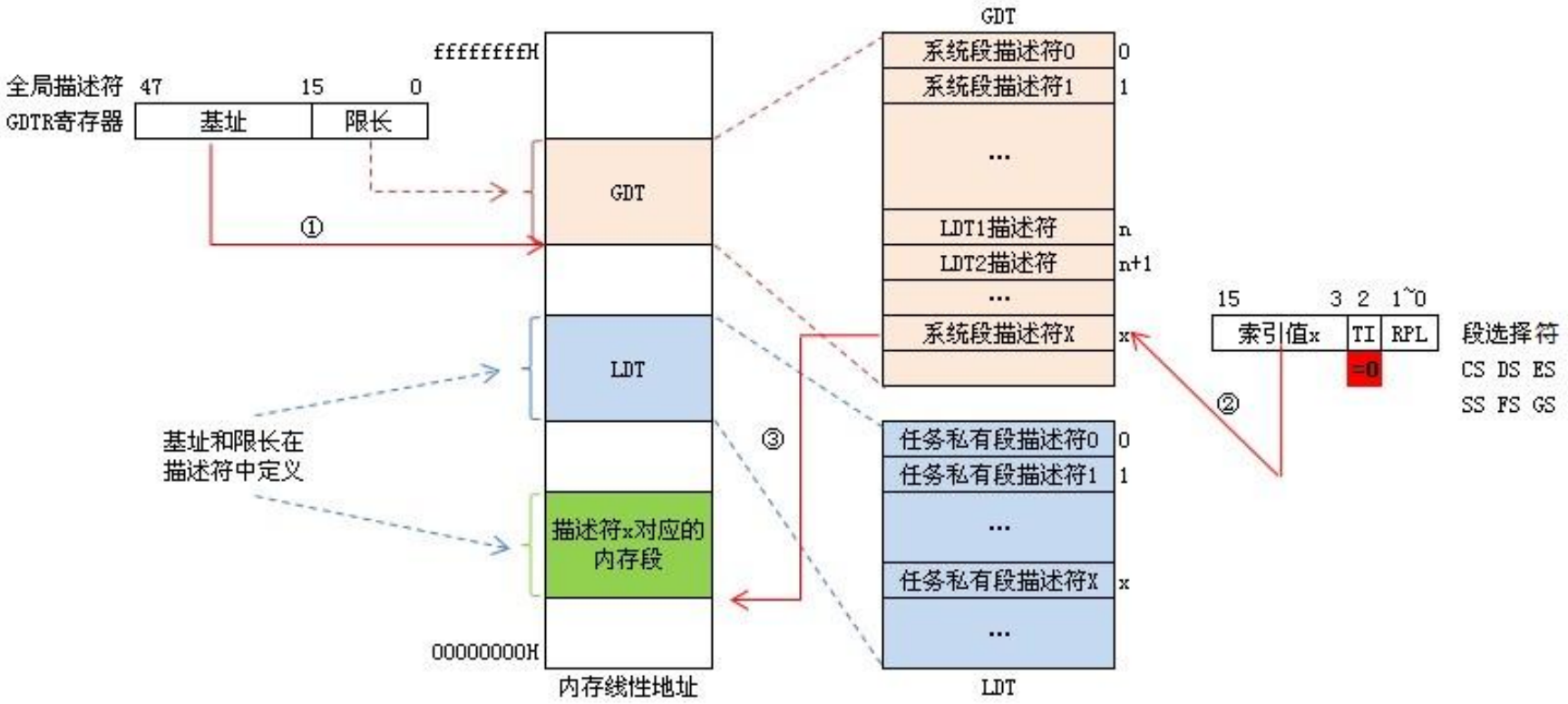
- 段寄存器
  - 每个段寄存器都有一个可见部分和一个隐藏部分
  - 当一个段选择符被加载到一个段寄存器的可见部分中，处理器同时也把相应段描述符中的段基地址、段限长及属性加载到段寄存器的隐藏部分
  - 在对描述符表中的描述符做任何改动之后，应立即重新加载段寄存器
  - 6个段寄存器
    - 代码段寄存器CS
    - 栈段寄存器SS
    - 数据段寄存器DS
    - 辅助的数据段寄存器ES、FS、GS



GDT与LDT


GDTR指向全局描述符表
TR指向状态段描述符
LDTR指向局部表段描述符
全局描述符表 GDT
局部表 ldtN
状态段 tssN
局部表 ldt1
状态段 tss1
局部表 ldt0
状态段 tss0
未用(syscall)
数据段 Data
代码段 Code
NULL
内核数据段
内核代码段
局部描述符表 LDT
数据段 Data
代码段 Code
NULL
任务 1
任务状态段
局部描述符表 LDT
数据段 Data
代码段 Code
NULL
任务 0
任务状态段
任务 1
用户数据
任务 1
用户代码
任务 0
用户数据
任务 0
用户代码
全局地址空间
局部地址空间

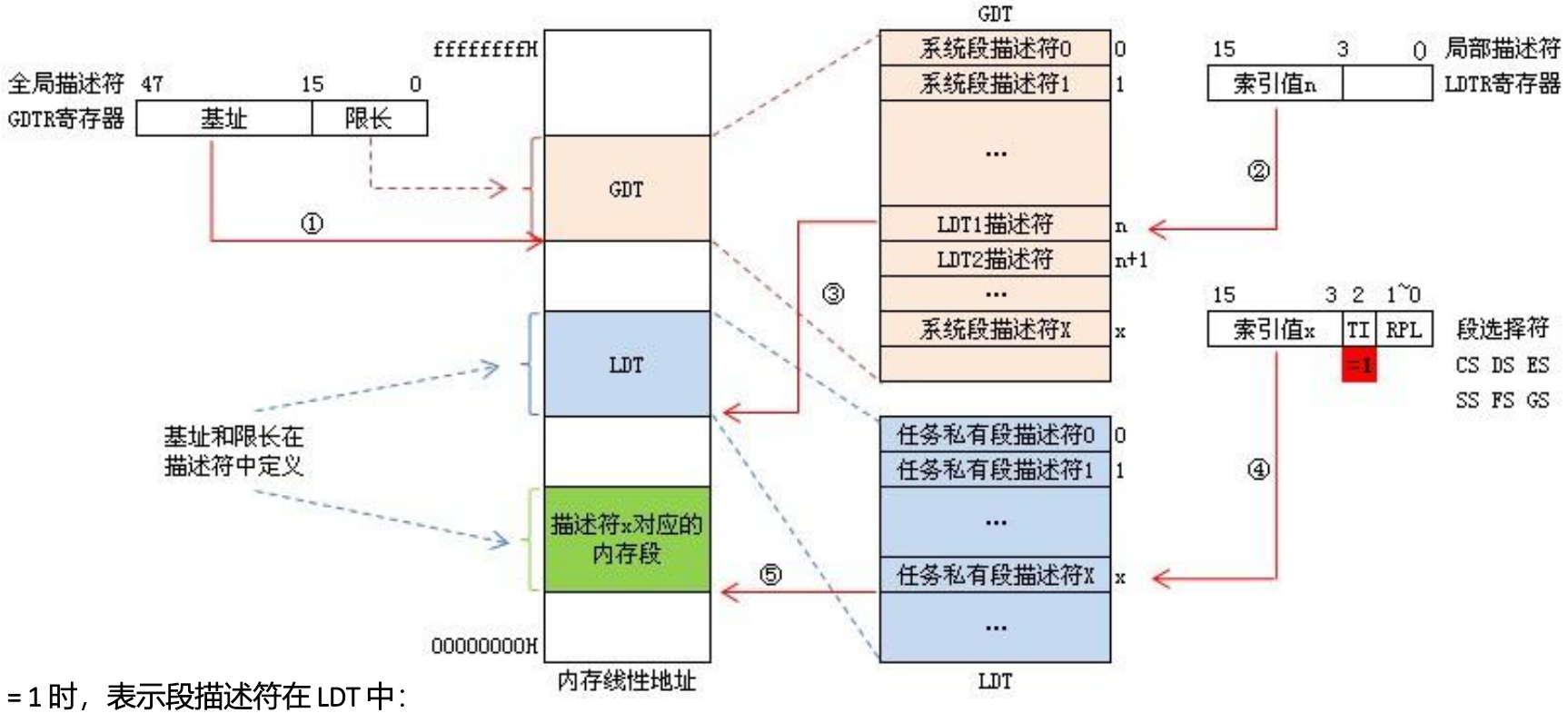
访问GDT



当 TI = 0 时，表示段描述符在 GDT 中：

1. 先从 GDTR 寄存器中获得 GDT 基址；
2. 然后再根据段选择子（段选择器）高13位的位置索引值得到段描述符；
3. 段描述符符包含段的基址、限长、优先级等各种属性，这就得到了段的起始地址（基址），再以基址加上偏移地址。

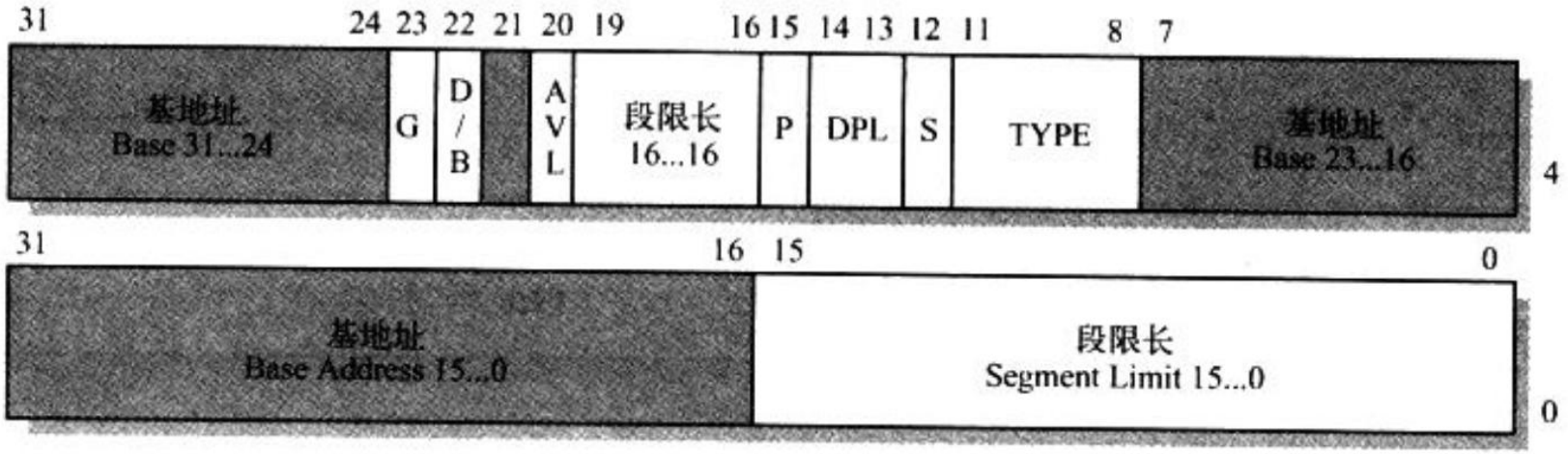
访问LDT



当 TI = 1 时，表示段描述符在 LDT 中：

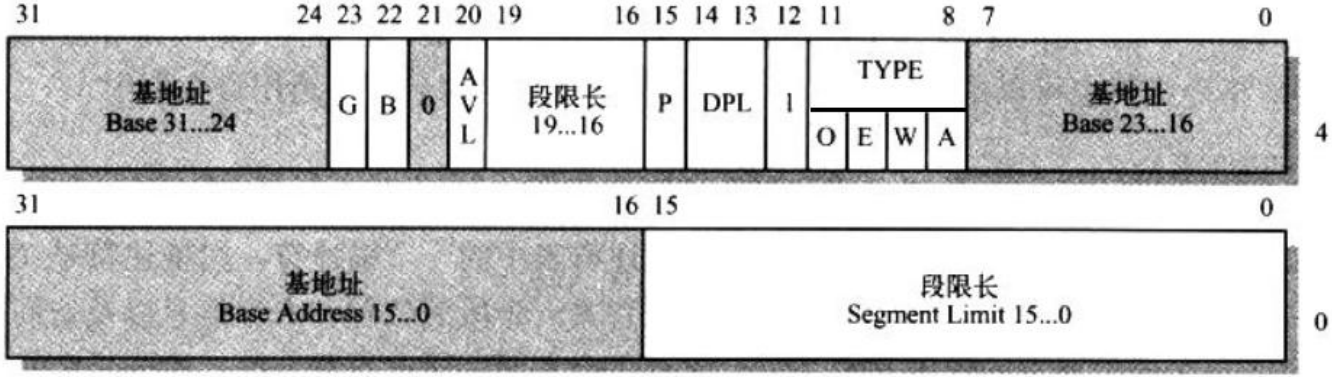
1. 还是先从 GDTR 寄存器中获得 GDT 基址；
2. 从 LDTR 寄存器中获取 LDT 所在段的位置索引 (LDTR高13位)，此时 LDTR 可以看成一个段选择子；
3. 通过这个位置索引 (LDTR高13位)，在 GDT 中得到 LDT 的段描述符从而得到 LDT 段基址；
4. 利用段选择子（段选择器）高13位位置索引值从 LDT 段中得到需要访问的内存的段描述符；
5. 段描述符符包含段的基址、限长、优先级等属性，这就得到了段的起始地址（基址），再以基址加上偏移地址得到最后的线性地址。

段描述符

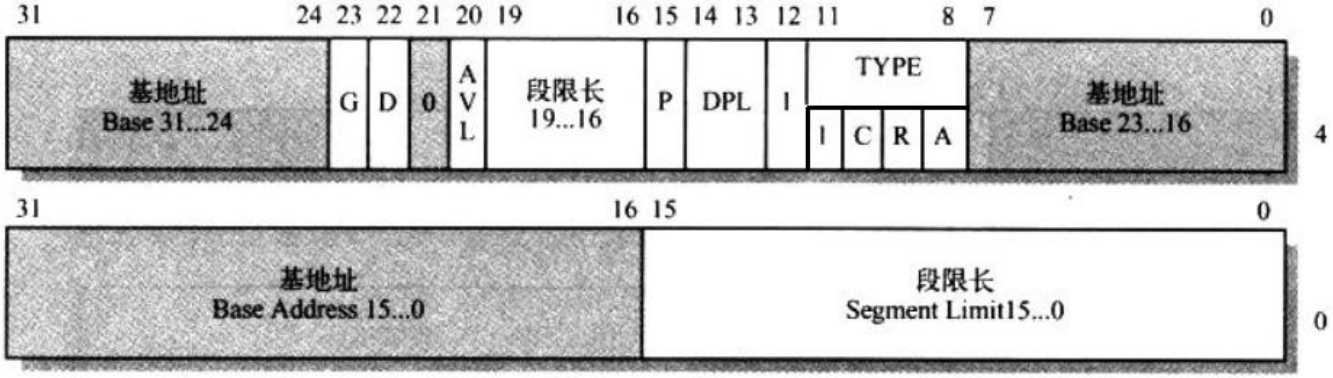


A	—— 已访问	D	—— 默认值	R	—— 可读
AVL	—— 软件可用位	DPL	—— 描述符特权级	LIMIT	—— 段限长
B	—— 默认大小	E	—— 扩展方向	W	—— 可写
C	—— 一致代码段	G	—— 颗粒度	P	—— 存在

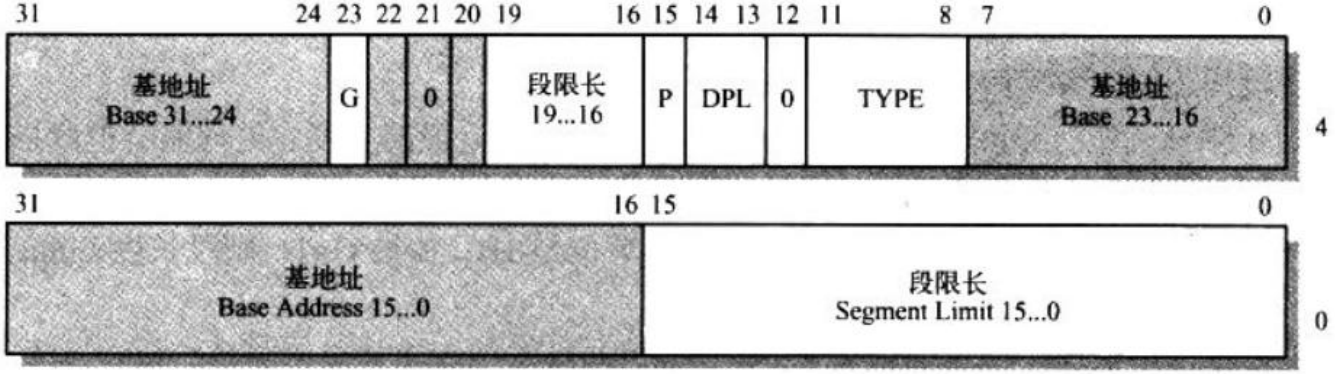
数据段描述符



代码段描述符



系统段描述符



代码段和数据段描述符类型

类型（TYPE）字段					描述符类型	说明
十进制	位 11	位 10	位 9	位 8		
		E	W	A		
0	0	0	0	0	数据	只读
1	0	0	0	1	数据	只读，已访问
2	0	0	1	0	数据	可读/写
3	0	0	1	1	数据	可读/写，已访问
4	0	1	0	0	数据	向下扩展，只读
5	0	1	0	1	数据	向下扩展，只读，已访问
6	0	1	1	0	数据	向下扩展，可读/写
7	0	1	1	1	数据	向下扩展，可读/写，已访问
		C	R	A		
8	1	0	0	0	代码	仅执行
9	1	0	0	1	代码	仅执行，已访问
10	1	0	1	0	代码	执行/可读
11	1	0	1	1	代码	执行/可读，已访问
12	1	1	0	0	代码	一致性段，仅执行
13	1	1	0	1	代码	一致性段，仅执行，已访问
14	1	1	1	0	代码	一致性段，执行/可读
15	1	1	1	1	代码	一致性段，执行/可读，已访问

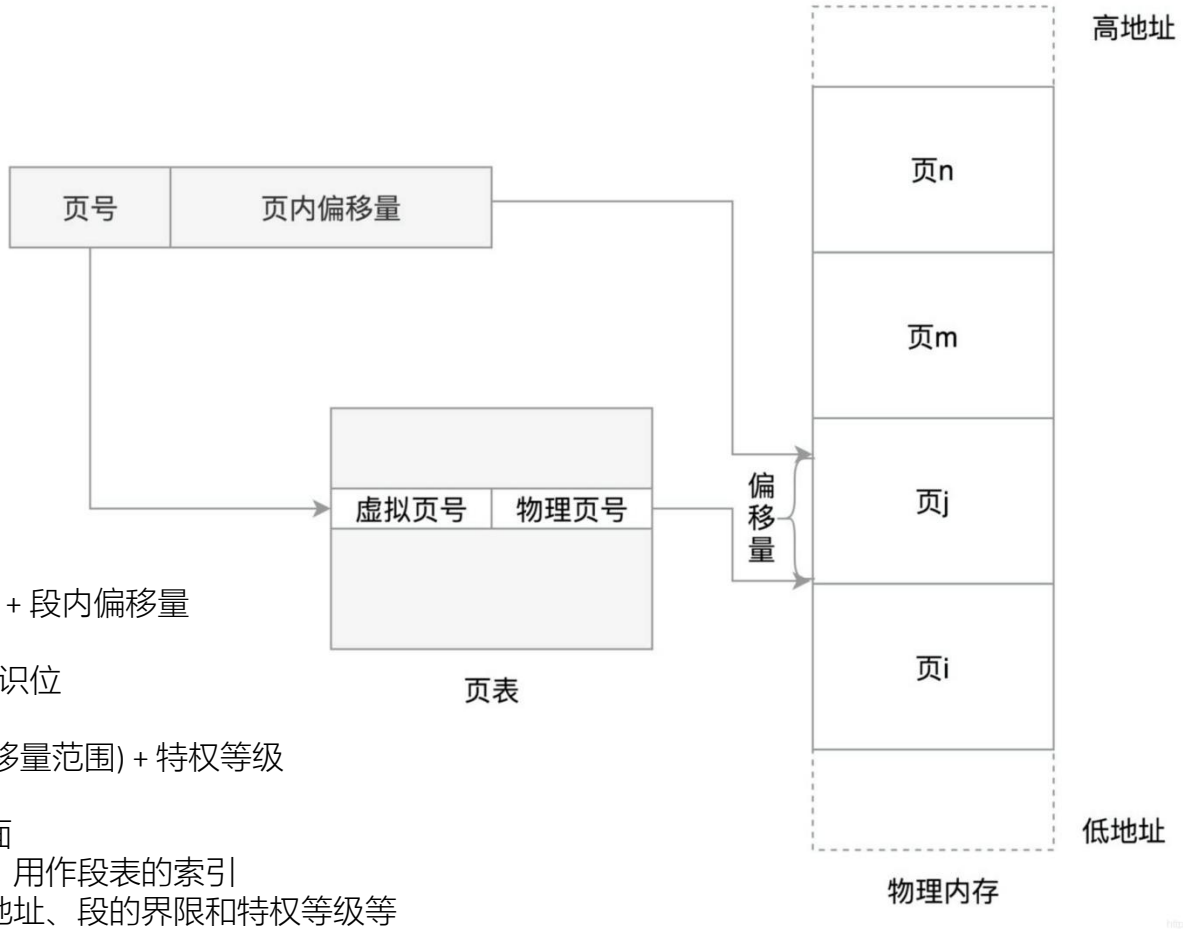
系统段描述符类型

类型（TYPE）字段					说明	
十进制	位 11	位 10	位 9	位 8		
0	0	0	0	0	Reserved	保留
1	0	0	0	1	16-Bit TSS (Available)	16 位 TSS（可用）
2	0	0	1	0	LDT	LDT
3	0	0	1	1	16-Bit TSS (Busy)	16 位 TSS（忙）
4	0	1	0	0	16-Bit Call Gate	16 位调用门
5	0	1	0	1	Task Gate	任务门
6	0	1	1	0	16-Bit Interrupt Gate	16 位中断门
7	0	1	1	1	16-Bit Trap Gate	16 位陷阱门
8	1	0	0	0	Reserved	保留
9	1	0	0	1	32-Bit TSS (Available)	32 位 TSS（可用）
10	1	0	1	0	Reserved	保留
11	1	0	1	1	32-Bit TSS (Busy)	32 位 TSS（忙）
12	1	1	0	0	32-Bit Call gate	32 位调用门
13	1	1	0	1	Reserved	保留
14	1	1	1	0	32-Bit Interrupt Gate	32 位中断门
15	1	1	1	1	32-Bit Trap Gate	32 位陷阱门

分段机制提供的保护

- 三个不同的特权级：
  - CPL（Current Privilege Level）：存放在代码段寄存器中，代表当前执行程序的特权级
  - RPL（Request Privilege Level）：请求特权级，存放在段选择子中
  - DPL（Descriptor Privilege Level）：请求特权级，存放在段选择子中
- 在保护模式下，CPU利用CPL/RPL/DPL对程序的访问操作进行权限检查
  - 数据段和堆栈段：只需要CPL <= 目标段DPL 且 RPL <= 目标段DPL，便可成功访问（即程序处于高特权级时可以访问相同或低特权级数据）

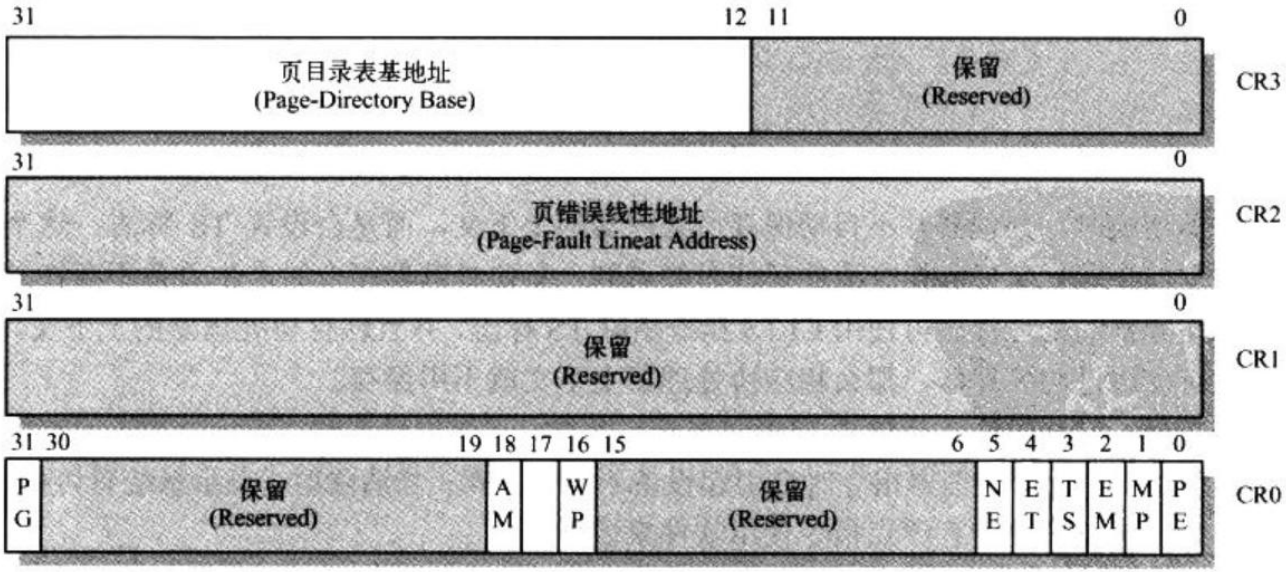
分页机制



分段机制下虚拟地址的组成

- 虚拟地址 = 段选择子(段寄存器) + 段内偏移量
- 段选择子 = 段号(段表索引) + 标识位
- 段表 = 物理基地址 + 段界限(偏移量范围) + 特权等级
- 段选择子：保存在段寄存器里面
  段选择子里面最重要的是段号，用作段表的索引
  段表里面保存的是这个段的基地址、段的界限和特权等级等
- 段内偏移量
  虚拟地址中的段内偏移量应该位于 0 和段界限之间
  段内偏移量是合法的，就将段基地址加上段内偏移量得到物理内存地址

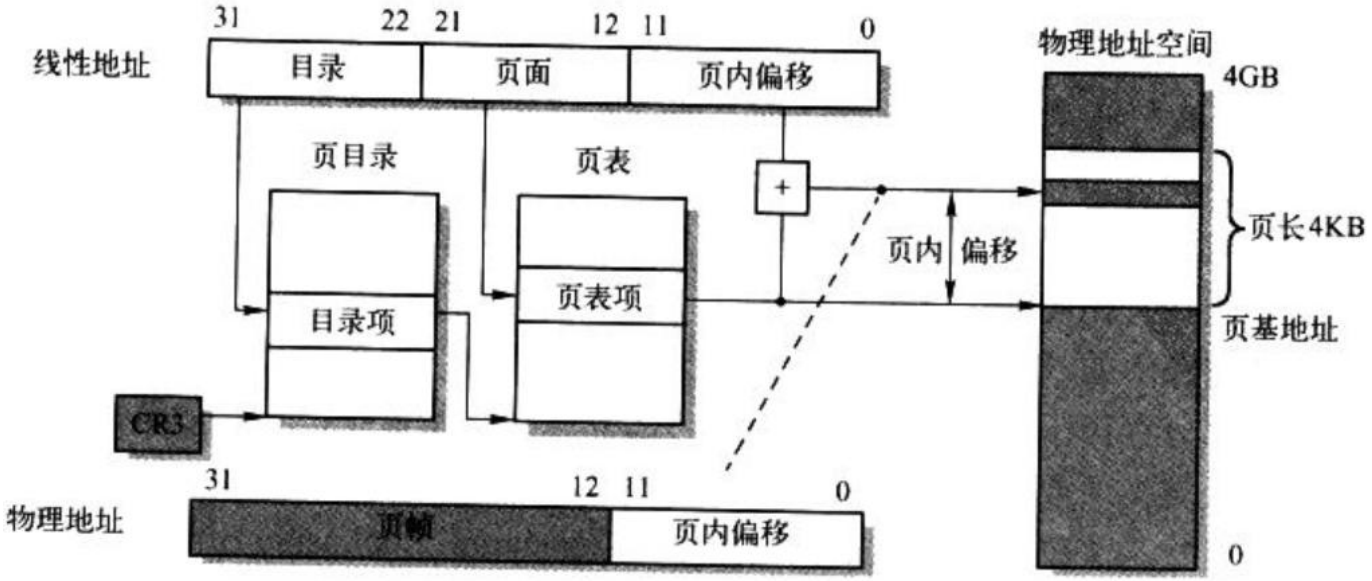
控制寄存器



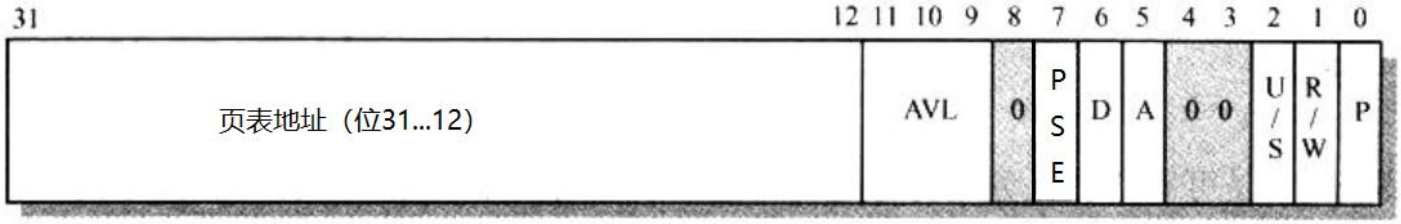
- CR0：含有控制处理器操作模式和状态的系统控制标志
- CR1：保留不用
- CR2：含有导致页错误的线性地址
- CR3：含有页目录表物理内存基地址
- PE：保护模式启用标志
- PG：分页机制启用标志
- WP：写保护标志（超级用户程序对用户级页面的写）
- NE：协处理器错误标志

两级页表结构

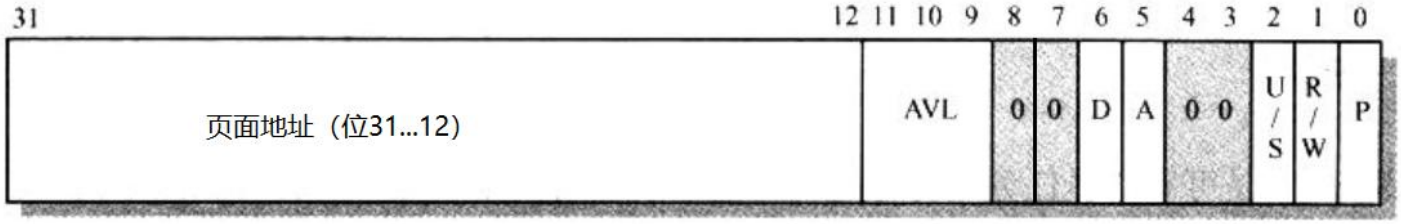
- 页表含有2^{20}（1M）个表项，每项4个字节。若作为一个表存放，最多将占用4MB内存。为减少内存占用，80x86 使用了两级页表。
- 第一级表称为页目录(page directory)，第二级表称为页表(page table)。
- 二级表结构允许页表被分散在内存各个页面中，并且不需要为不存在的或线性地址空间未使用的部分分配二级页表。虽然目录表页面必须总是存在于物理内存中，但是级页表可以在需要时再分配。



页目录项和页表项的格式



a) 页目录项格式



b) 页表项格式

- P：存在标志，P=1表示该表项指向的页在内存中，P=0表示该表项指向的页不在内存中
- R/W：读写标志
- U/S：管理员/用户标志
- A：访问标志
- PSE：页大小标志，如果置为1则表明页目录项指向的是4MB的页面，置为0则为4KB

Linux对分段机制的使用

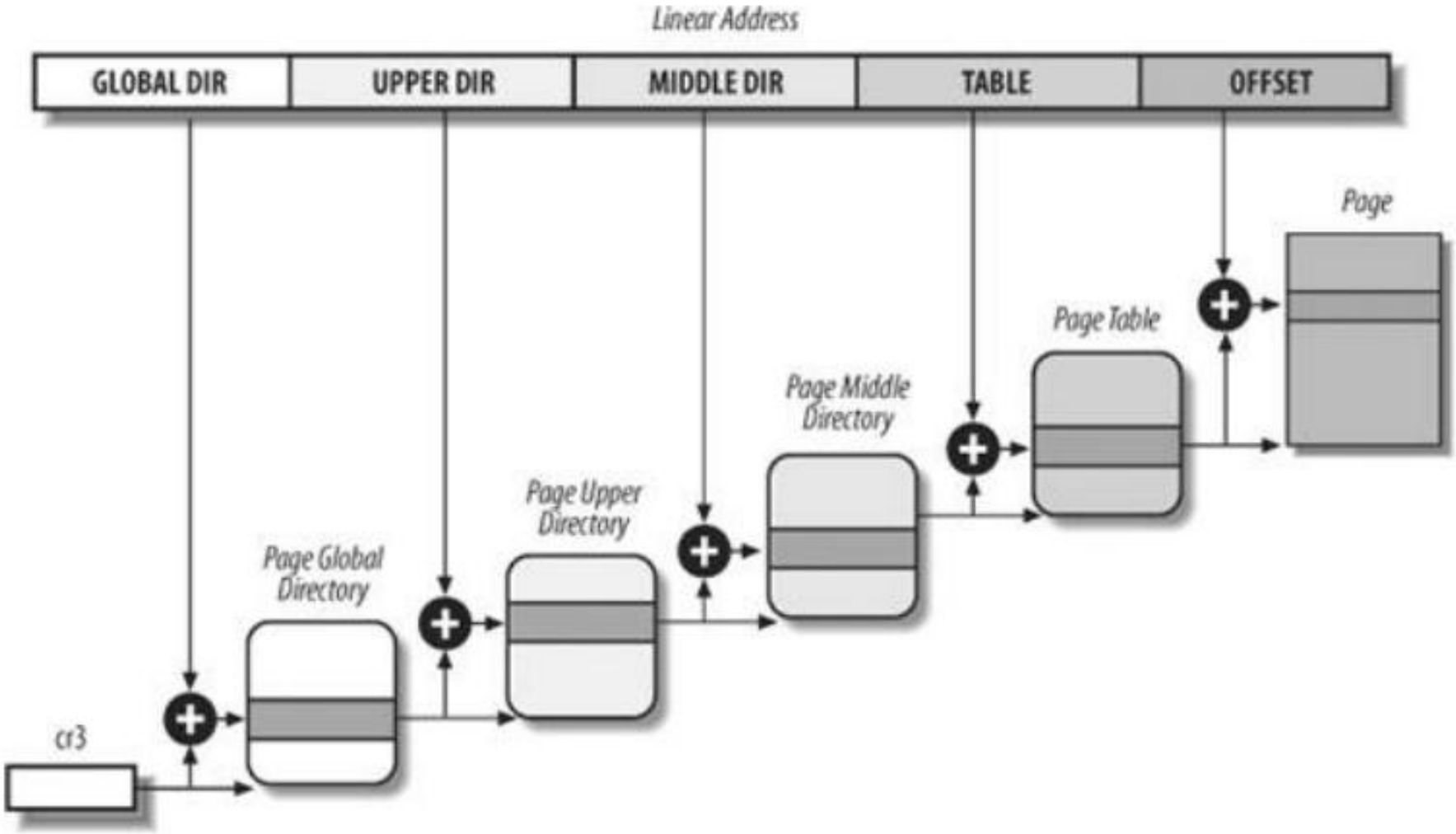
- Linux内核设计并没有全部采用Intel所提供的段方案，仅有限度地使用了分段机制，目的是简化Linux内核设计以及提高可移植性
  - Linux内核不区分数据段和栈段
  - Linux的进程没有使用LDT，对TSS的使用也非常有限，每个CPU一个TSS
  - 定义了4个段选择符：
    - 内核代码段选择符__KERNEL_CS：TI=0，RPL=0
    - 内核数据段描述符__KERNEL_DS：TI=0，RPL=0
    - 用户代码段选择符__USER_CS：TI=0，RPL=3
    - 用户数据段描述符__USER_DS：TI=0，RPL=3

```
/*
 * The layout of the per-CPU GDT under Linux:
 *
 *   0 - null
 *   ------- start of kernel segments:
 *
 *  12 - kernel code segment
 *  13 - kernel data segment
 *  14 - default user CS
 *  15 - default user DS
 */
```

```
#define GDT_ENTRY_INVALID_SEG       0
#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)
```

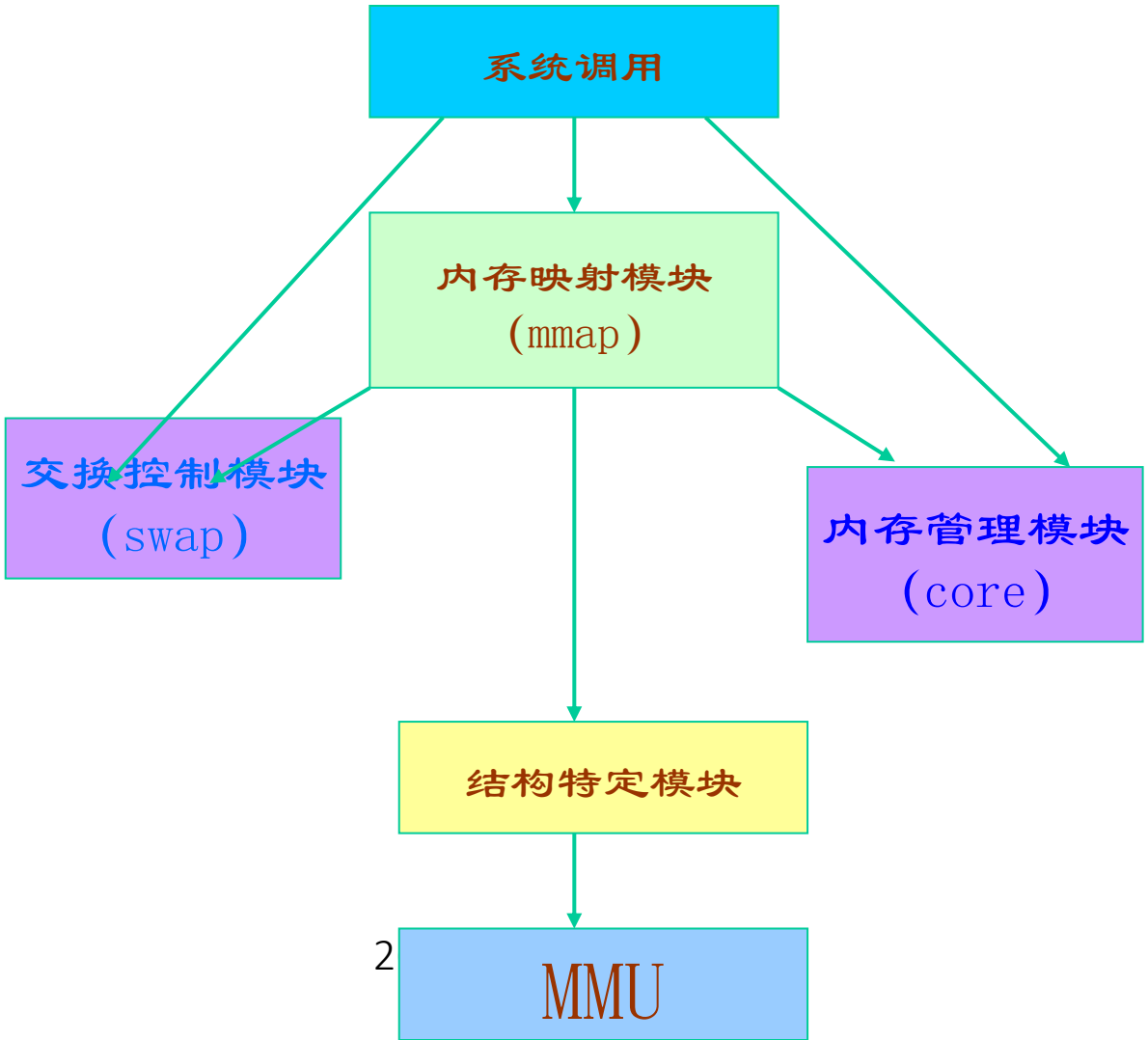
Linux对分页机制的使用



Linux

步骤	用户态 → 内核态	内核态 → 用户态
1. 硬件触发条件	中断/异常/系统调用（如 int 0x80 或 syscall）	执行 iret 指令（从内核返回用户态）
2. TSS 作用	**从当前 CPU 的 TSS 中读取 esp0（内核栈指针）**	**无 TSS 修改**，仅依赖栈中保存的用户态上下文
3. 栈切换	硬件自动切换到内核栈（ss=内核段, esp=esp0）	硬件从内核栈弹出用户态 ss/esp，恢复用户栈
4. 上下文保存	硬件压入用户态 ss/esp/eflags/cs/eip 到内核栈	硬件从内核栈弹出用户态 eip/cs/eflags/esp/ss
5. 段寄存器更新	cs/ss 切换为内核段选择子（DPL=0）	cs/ss 恢复为用户段选择子（DPL=3）
6. 特权级变化	CPL（当前特权级）从 3→0	CPL 从 0→3
7. 分页机制协同	CR3 不变（共享页表，内核空间标记为 Supervisor）	CR3 不变（仍指向进程页表，用户空间标记为 User）
8. 关键寄存器变化	eip 指向内核入口点（如系统调用处理函数）	eip 指向用户态返回地址（如系统调用后的下一条指令）

Linux的存储管理思想



Linux的存储管理思想

✓说明:

上图是虚拟内存管理的程序模块，实现代码大部分放在/mm目录下。

- 内存映射模块（mmap）

  负责把磁盘文件或交换空间文件的逻辑地址映射到虚拟地址，以及把虚拟地址映射到物理地址。

Linux的存储管理思想

- 交换模块（swap）

  负责控制内存内容的换入和换出。采用交换机制，从主存中淘汰最近没被访问的逻辑页，保存近来访问过的逻辑页。

- 核心内存管理模块（core）

  负责核心内存管理功能，如页的分配、回收和请求调页处理等功能，这些功能将别的内核子系统（如文件系统）所使用。

Linux的存储管理思想

- 结构特定的模块

  负责给各种硬件平台提供通用接口，主要完成主存初始化工作及对页面故障的处理。这个模块是实现虚拟内存的物理基础。

物理内存空间管理

- 数据结构
- 基于Buddy算法的内存页面管理
- 基于slab算法的内存分区管理

物理内存空间管理

- 数据结构
  - ✓ 分页管理结构
  - ✓ 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：

```
typedef struct page {
            struct page *next;
            struct page *prev;
            struct inode *inode;
            unsigned long offset;
            struct page *next_hash;
            atomic_t count;
            unsigned flags;
            unsigned dirty,age;
            struct wait_queue  *wait;
            struct buffer_head * buffers;
            unsigned long swap_unlock_entry;
            unsigned long map_nr;
} mem_map_t;
```

物理内存空间管理

- ✓ 说明:
  - Count
    共享该页面的进程计数
  - Age
    标志页面的"年龄"
  - Dirty
    表示该页面是否被修改过
  - prev和next:
    把page结构体链接成一个双向循环链表
  - prev_hash和next_hash
    把有关page结构体连成哈希表

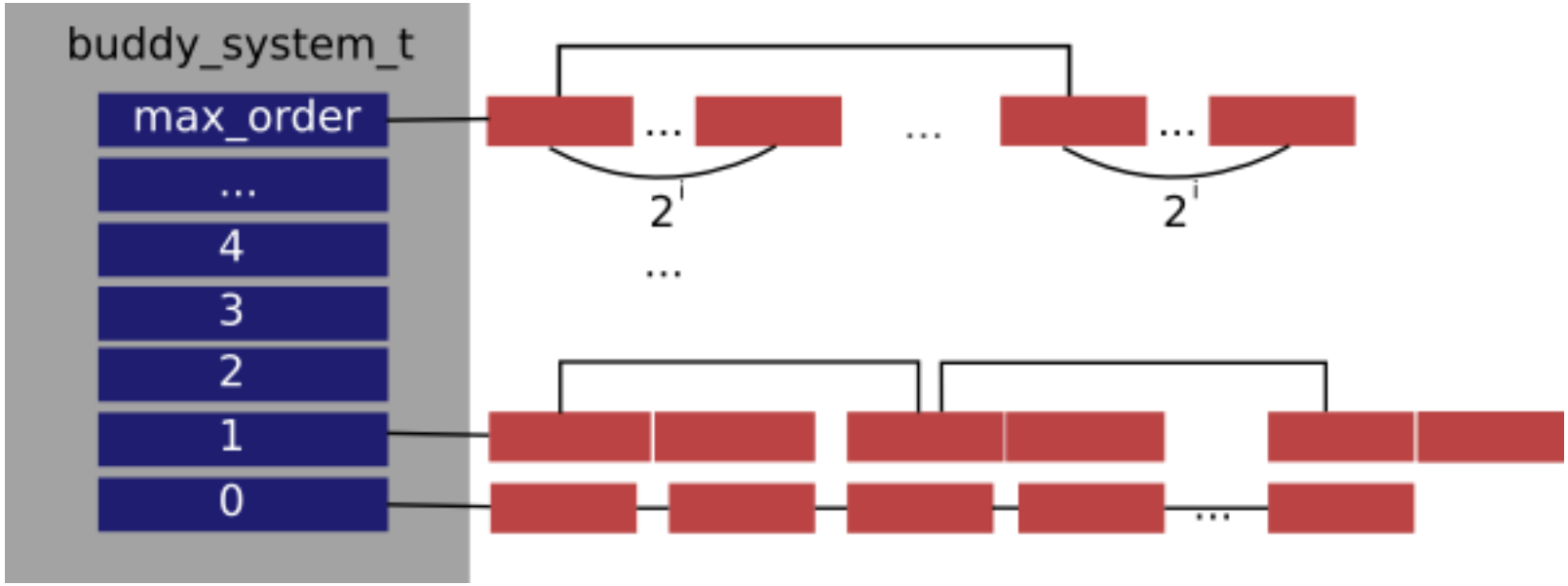
物理内存空间管理

- inode和offset：
  内核以节点和其上的偏移为键值，将页面组织成哈希表。
- Wait
  等待该页资源的进程等待队列指针
- Flag
  页面标志
- 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。

物理内存空间管理

- Buddy算法
  - ✓ **内存划分**：把内存中所有页面按照2^n划分，其中n=0~5，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个页面进行六次划分。
  - ✓ **链表数组管理**：维护一个链表数组，索引为 i 的链表存储所有大小为 2^i的未分配块（i 称为块的阶）。
  - ✓ **合并（Coalesce）**：若链表 i 中存在两个相邻且大小相等的空闲块（即"伙伴"），则合并为 2^{i+1}大小的块，并加入链表 i+1（需满足自然对齐）。
  - ✓ **分裂（Split）**：若请求分配 2^i大小的块时链表i为空，则从链表i+1中分配更大块，递归分裂直至满足需求。

物理内存空间管理



物理内存空间管理

内存划分规则

- **划分单位**：以物理页为基本单位（例如每页4KB），将内存按2的幂次方划分为不同大小的块。
- 阶数范围：通常支持6个阶（n=0~5），对应块大小为：

阶数 (i)	块大小（页面数）	块大小（假设页=4KB）
0	1	4KB
1	2	8KB
2	4	16KB
3	8	32KB
4	16	64KB
5	32	128KB

- 自然对齐：每个块的起始地址必须是其大小的整数倍。
- 示例：阶3的块（8页）起始地址需对齐到8页×4KB=32KB边界（如0x8000、0x10000等）。

物理内存空间管理

- 链表数组管理
  - 数据结构：维护一个空闲链表数组free_area[]，数组索引对应阶数i，每个链表存储该阶所有空闲块。

```
struct free_area {
    struct list_head free_list;    // 空闲块链表头
    unsigned long nr_free;         // 空闲块数量
};
```

  - 块表示：每个空闲块的首个物理页元数据中记录阶数，并通过链表指针串联同阶块。
  - 示例：阶2链表中的每个块大小为4页，首个页的struct page中存储order=2。

物理内存空间管理

- 合并流程
  - 伙伴条件：两个相邻同阶块必须满足：
    - 地址连续且自然对齐到2^{i+1}大小边界。
    - 均处于空闲状态。
  - 合并步骤：
    1. 从链表i中移除两个伙伴块。
    2. 合并为阶i+1的块，计算新块起始地址（例如块A地址为P，伙伴块B地址为P + 2^i）。
    3. 将合并后的块加入链表i+1。
    - 示例：阶1的两个块（地址0x8000和0x9000）无法合并（未对齐到16KB），但地址0x8000和0xC000的块可合并为阶2块。

物理内存空间管理

- 分裂流程
  - 触发条件：分配阶i的块时，链表i为空，需从更高阶链表i+1分配。
  - 递归分裂：
    1. 从链表i+1取出一个块，标记为已分配。
    2. 将其分裂为两个阶i的伙伴块。
    3. 将其中一个块分配给请求，另一个加入链表i。
    - 示例：请求分配阶2（4页）的块：
      - 若链表2为空，从链表3（8页块）取一块。
      - 分裂为两个4页块（地址P和P+4页），分别加入链表2。
      - 分配地址P的块，剩余块P+4页保留在链表2。

物理内存空间管理

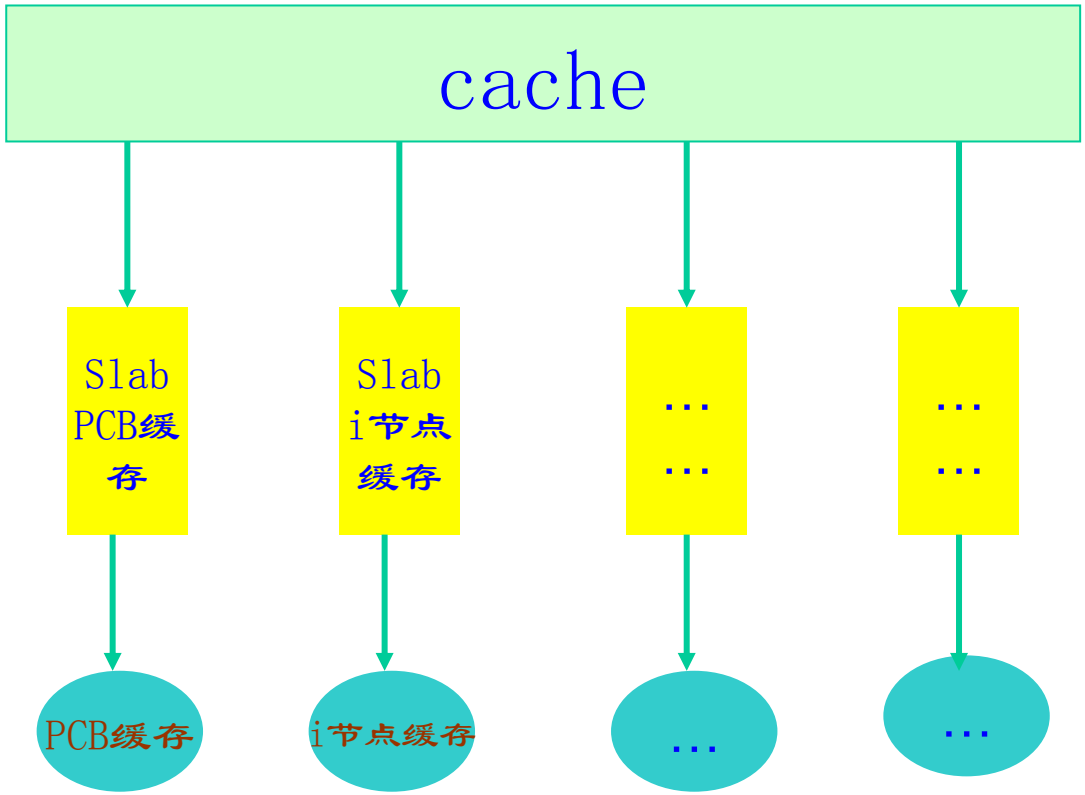
Slab算法

- 与buddy system协同
  - Slab初始化：Slab缓存从伙伴系统申请大块内存（如阶5的32页块）。
  - 对象分配：Slab将大块划分为多个小对象（如task_struct），管理对象复用。
  - 内存返还：当Slab缓存空闲对象过多时，将大块返还伙伴系统，减少内存占用。
- 优化场景：高频小对象（如task_struct、inode）的分配/释放性能。
- 核心思想：预分配同类型对象的连续内存区域（称为Slab），减少内部碎片。

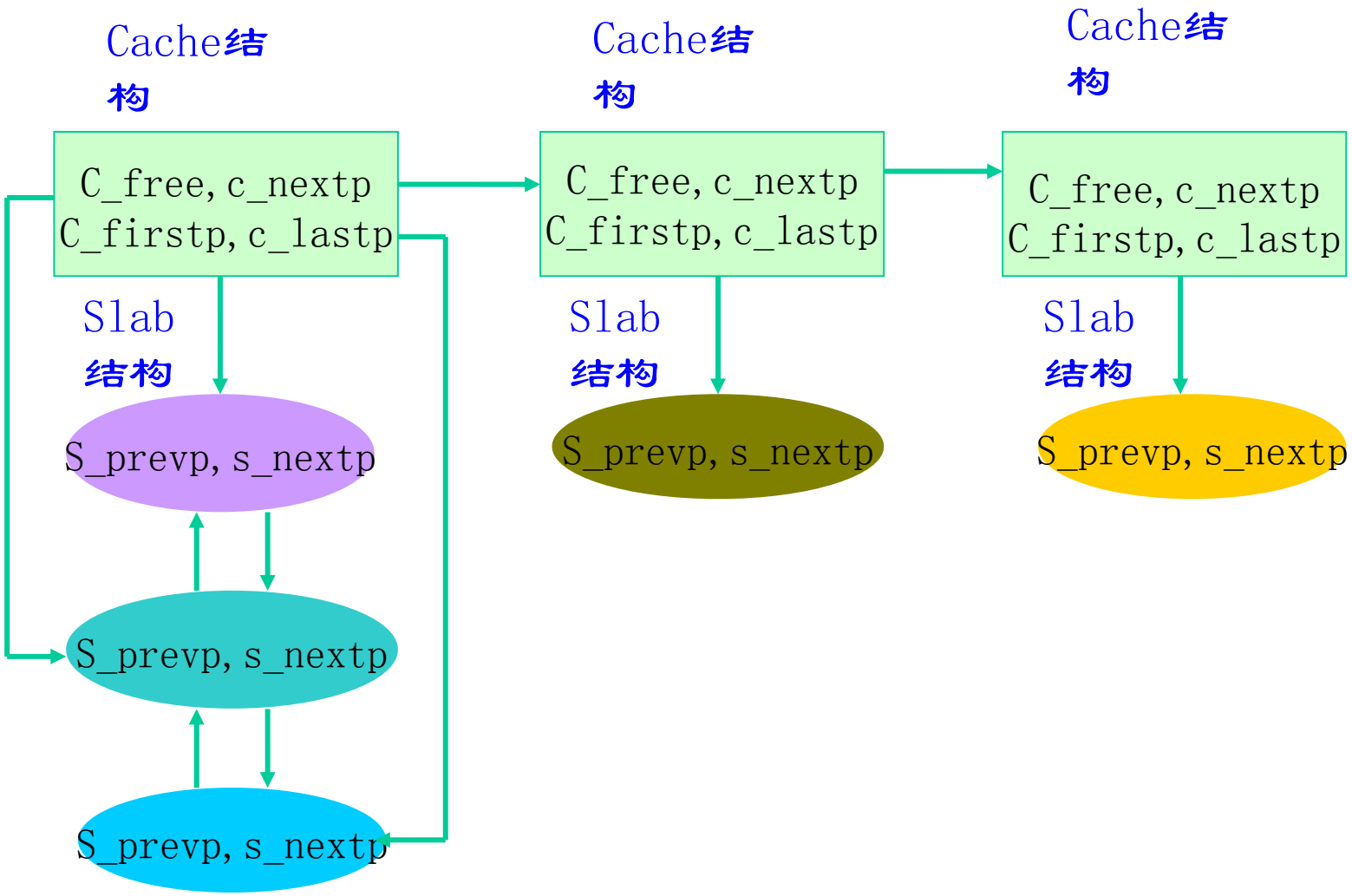
物理内存空间管理

核心组件

组件	描述
Slab缓存（Slab Cache）	管理同一类型对象的多个Slab，每个Slab包含多个等大小的对象。
Slab	由连续物理页组成（通常来自伙伴系统），划分为多个对象（空闲/已分配）。
Magazine	处理器私有缓存（避免全局锁竞争），每个CPU维护两个Magazine（当前和备用）。
全局Full Magazine列表	存储所有处理器的已满Magazine，供其他处理器按需获取。



Slab分配器的组成



Cache结构与slab结构之间的关系

物理内存空间管理

对象生命周期

- 初始化：首次分配时，从Slab中取对象并调用构造函数（初始化对象状态）。
- 使用：对象被内核模块使用。
- 释放：
  - 对象放入当前CPU的Magazine（不立即释放，便于快速重用）。
  - Magazine满时，移至全局Full Magazine列表。
- 回收：
  - 内存不足时，从全局列表回收对象，调用析构函数并返还Slab。
  - 空Slab立即释放回伙伴系统。

物理内存空间管理

分配流程（slab_alloc()）

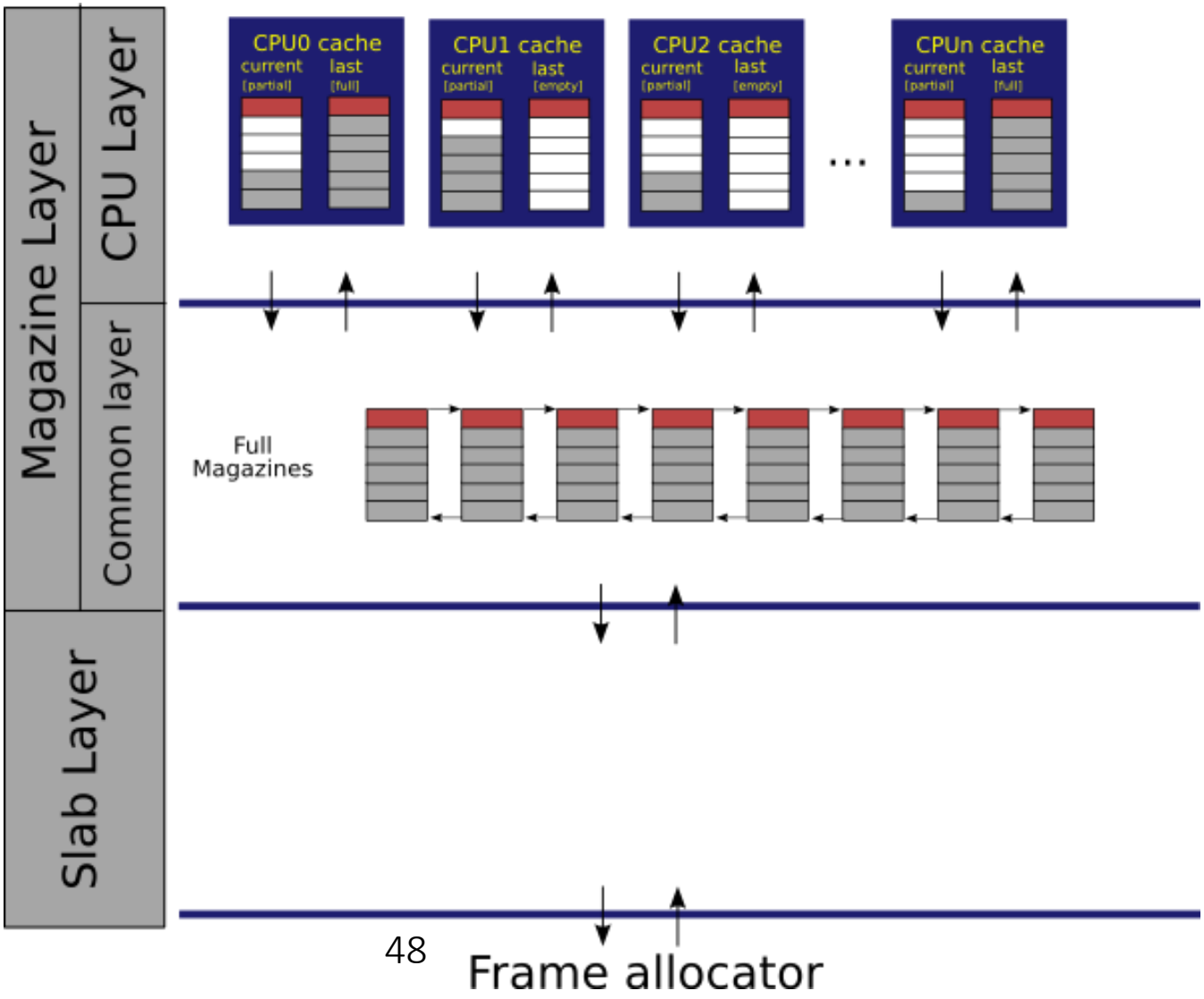
1. 检查当前CPU的Magazine是否有空闲对象，若有则直接返回。
2. 若当前Magazine为空，切换至备用Magazine并重试。
3. 若两个Magazine均空，从全局Full Magazine列表加载一个Magazine到当前CPU。
4. 若全局列表为空，直接从Slab分配新对象（触发Slab扩容或伙伴系统申请）。

物理内存空间管理

释放流程（slab_free()）

1. 若当前CPU的Magazine未满，将对象放入其中。
2. 若当前Magazine已满，切换至备用Magazine并重试。
3. 若两个Magazine均满，申请新Magazine，将满Magazine移至全局列表。
4. 内存不足时，直接释放对象到Slab（无需额外内存分配）。

物理内存空间管理


Magazine Layer
CPU Layer
Common layer
Slab Layer
CPU0 cache
current
[partial]
last
[full]
CPU1 cache
current
[partial]
last
[empty]
CPU2 cache
current
[partial]
last
[empty]
...
CPUn cache
current
[partial]
last
[full]
Full
Magazines
Frame allocator

虚拟地址空间管理

- 进程地址空间
- 虚拟内存空间管理（内存映射）

进程地址空间

- Linux把进程虚拟空间分成两部分：内核区和用户区
- 操作系统内核的代码和数据等被映射到内核区。进程可执行映像（代码和 数据）映射到虚拟内存的用户区。

进程地址空间

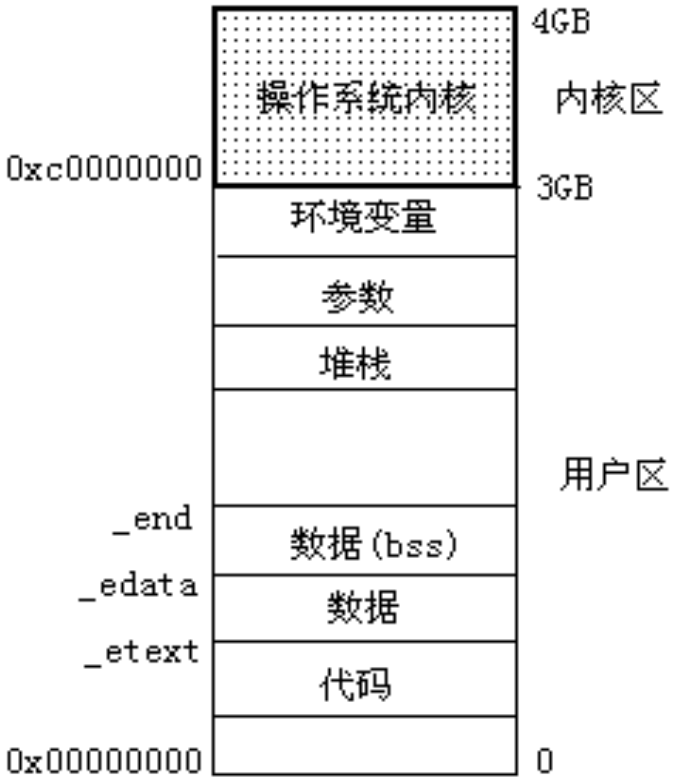
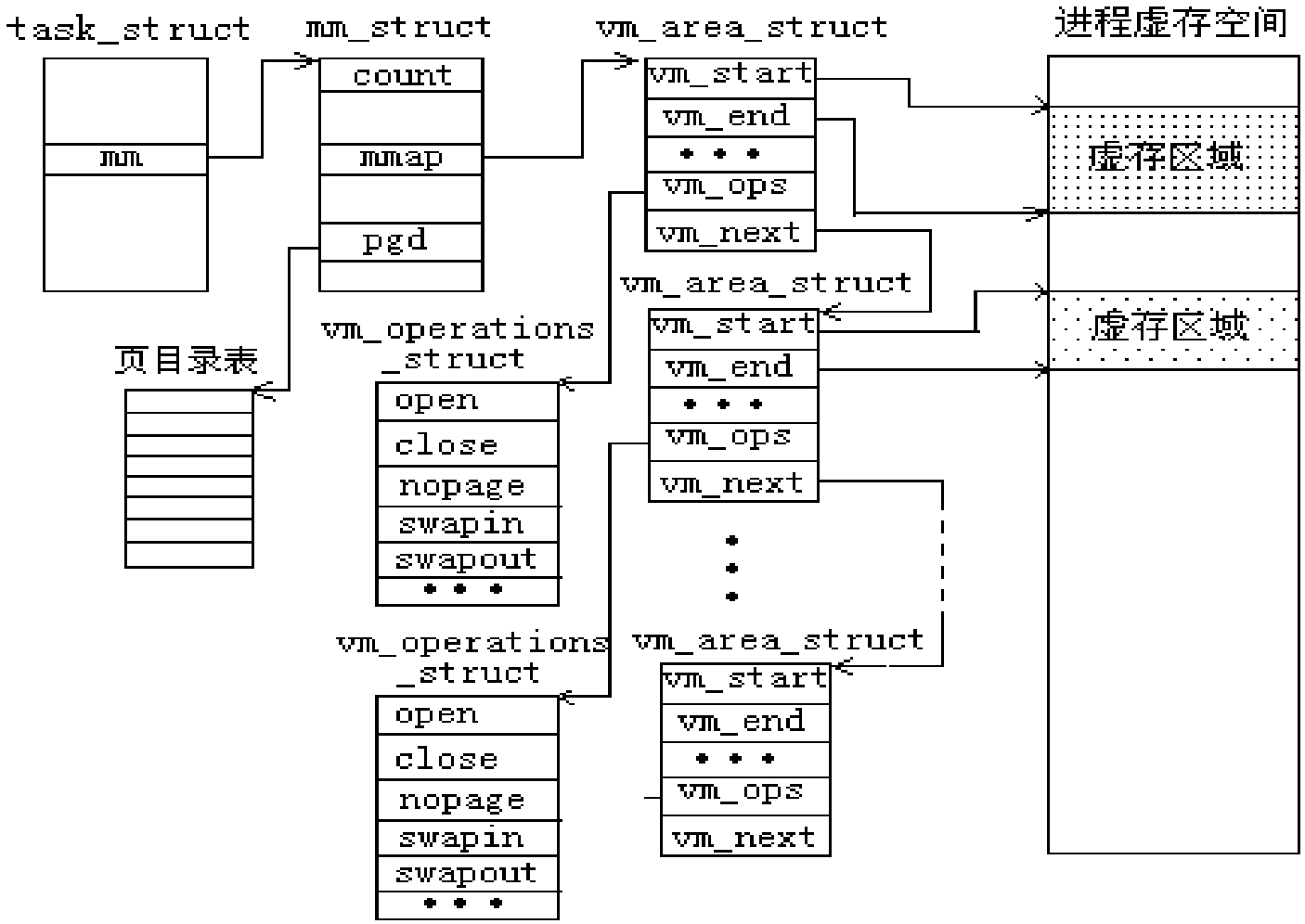


图5.1 进程的虚存空间

虚拟空间管理



进程虚存空间管理

虚拟空间管理

- 有关数据结构
  - ✓mm_struct结构体

定义了每个进程的虚存用户区，首地址在任务结构体中，定义在/include/linux/schedul.h中。

```
struct mm_struct {
        int count;
        pgd_t * pgd;
        unsigned long context;
        unsigned long start_code, end_code,
start_data, end_data;
        unsigned long start_brk, brk, start_stack,
start_mmap;
        unsigned long arg_start, arg_end, env_start,
env_end;
        unsigned long rss, total_vm, locked_vm;
        unsigned long def_flags;
        struct vm_area_struct * mmap;
        struct vm_area_struct * mmap_avl;
        struct semaphore mmap_sem;
};
```

虚拟空间管理

✓进程虚存区域

一个虚存区域是虚存空间中一个连续区域，每个虚拟区域用一个vm_area_struct结构体描述，定义在/include/linux/mm.h中。

```
struct vm_area_struct {
      struct mm_struct * vm_mm;
      unsigned long vm_start;
      unsigned long vm_end;
      pgprot_t vm_page_prot;
      unsigned short vm_flags;
      short vm_avl_height;
            struct vm_area_struct * vm_avl_left;
            struct vm_area_struct * vm_avl_right;
            struct vm_area_struct * vm_next;
            struct vm_area_struct * vm_next_share;
            struct vm_area_struct * vm_prev_share;
            struct vm_operations_struct * vm_ops;
            unsigned long vm_offset;
            struct inode * vm_inode;
            unsigned long vm_pte;
};
```

虚拟空间管理

✓ 说明:

- vm_mm指针指向进程的mm_struct结构体
- vm_start和vm_end 虚拟区域的开始和终止地址。
- vm_page_prot　虚存区域的页面的保护特性
- vm_inode

  若虚存区域映射的是磁盘文件或设备文件内容，则vm_inode指向这个文件的inode结构体，否则vm_inode为NULL。

虚拟空间管理

- vm_flags指出了虚存区域的操作特性:

VM_READ	虚存区域允许读取
VM_WRITE	虚存区域允许写入
VM_EXEC	虚存区域允许执行
VM_SHARED	虚存区域允许多个进程共享
VM_GROWSDOWN	虚存区域可以向下延伸
VM_GROWSUP	虚存区域可以向上延伸
VM_SHM	虚存区域是共享存储器的一部分
VM_LOCKED	虚存区域可以加锁
VM_STACK_FLAGS	虚存区域做为堆栈使用

虚拟空间管理

- vm_offset

  该区域的内容相对于文件起始位置的偏移量，或相对于共享内存首址的偏移量。

- vm_next

  所有vm_area_struct结构体链接成一个单向链表

  vm_next指向下一个vm_area_struct结构体。链表首地址由mm_struct中成员项mmap指出。

- vm_ops是指向vm_operations_struct结构体的指针，该结构体中包含着指向各种操作函数的指针。

虚拟空间管理

- vm_avl_left
  左指针指向相邻的低地址虚存区域
- vm_avl_right
  右指针指向相邻的高地址虚存区域
- mmap_avl
  表示进程AVL树的根
- vm_avl_hight
  表示AVL树的高度。
- vm_next_share和vm_prev_share

把有关vm_area_struct结合成一个共享内存时使用的双向链表。

虚拟空间管理

➢虚存区域建立

- ✓ Linux使用do_mmap()函数完成可执行映像向虚存区域的映射，建立有关的虚存区域。
- ✓ do_mmap()函数定义在/mm/mmap.c文件中

```
unsigned long do_mmap(struct file * file,
unsigned long addr,
        unsigned long len,
        unsigned long prot,
        unsigned long flags,
        unsigned long off)
```

addr　虚存区域在虚拟内存空间的开始地址

len　是这个虚存区域的长度。

file　是指向该文件结构体的指针

off　是相对于文件起始位置的偏移量。

虚拟空间管理

prot 指定了虚存区域的访问特性:

PROT_READ	0x1	对虚存区域允许读取
PROT_WEITE	0x2	对虚存区域允许写入
PROT_EXEC	0x4	虚存区域（代码）允许执行
PROT_NONE	0x0	不允许访问该虚存区域

flag 指定了虚存区域的属性:

MAP_FIXED	指定虚存区域固定在addr的位置上
MAP_SHARED	指定对虚存区域的操作是作用在共享页面上
MAP_PRIVATE	指定了对虚存区域的写入操作将引起页面拷贝

请页机制

- 缺页中断处理函数

  do_page_fault()

触发缺页的虚拟地址
(Fault Address)

异常状态寄存器
(Exception Syndrome Register)

```
static int __kprobes do_page_fault(unsigned long far, unsigned long esr,
                                   struct pt_regs *regs)
{
        const struct fault_info *inf;
        struct mm_struct *mm = current->mm;
        vm_fault_t fault;
        unsigned long vm_flags;
        unsigned int mm_flags = FAULT_FLAG_DEFAULT;
        unsigned long addr = untagged_addr(far);
        struct vm_area_struct *vma;
        int si_code;
        int pkey = -1;

        if (kprobe_page_fault(regs, esr))
                return 0;
```

内核栈中保存的寄存器上下文
(Processor Registers)

交换机制

- 引入目的

  利用外存解决物理内存存储量不足问题

- 方法

  将磁盘上一部分空间作为交换空间，当物理内存不足时，将一部分暂不用的数据换入到交换空间中，从而保证内存有足够空间空间

- 优缺点

  解决了主存不足问题，增加了处理机开销

交换机制

- 交换空间
- 交换空间格式
- 守护进程kswapd（）

交换机制

- 交换空间
  - ✓ 交换设备和交换文件统称为交换空间
  - ✓ 交换设备中同一个页面中数据块连续存放
  - ✓ 交换文件中是零散存放的，需要通过交换文件索引点来检索。

交换机制

交换空间格式

- ✓ 交换空间被划分为块（页插槽），每个块等于一个物理页。第一个页插槽中，存放了一个以"swap_space"结尾的位图，位图每一位对应一个页插槽，即第一个插槽后是真正用于交换的页插槽，这样，每个交换空间可容纳：

  (4096 - 10) ×8 - 1 = 32687个页面

- ✓ Linux允许建立8个交换空间

交换机制

- 页面交换守护进程kswapd（）
  - ✓ 是一个无限循环的线程，完成页面交换工作，保证系统内有足够内存。
  - ✓ Linux将页面分为活跃状态、非活跃”脏“状态、非活跃”干净“状态。需要时，将非活跃”脏“状态页面内容写入到外存交换空间，并将该页面从”脏“队列移动到”干净“队列。回收页面总是从非活跃”干净“链表队列中进行

缓冲机制

- 目的

  改善处理机和外围设备之间速度不匹配的矛盾，提高系统性能。

- Linux采用了多种与主存相关的高速缓存。
  - 缓冲区高速缓存
  - 页面高速缓存
  - 交换高速缓存
  - 硬件高速缓存

缓冲机制

- 缓冲区高速缓存
  - 大小固定，是针对块设备的I/O开辟的。
  - 缓存区高速缓存以块设备标识符和块号作为索引，快速查找数据块，若数据块已在缓存中，则不必从物理设备中读取，从而加快了读取数据速度。
  - 缓冲区高速缓存大小可变化，当没有空闲的缓冲区而又必须分配新缓冲区时，内核就按需分配缓冲区，当主存空间不足时，可以释放缓冲区。

缓冲机制

页面高速缓存

- 是为了加快对磁盘文件访问而设立的。
- 当页面从磁盘被读入主存时，被存入页面高速缓存。
- 如文件系统的一些系统调用，如read()、write()等，都是通过页面高速缓存完成的。

缓冲机制

- 交换缓存
  - 为了避免页面交换时，直接对磁盘交换空间操作，从而提高了系统性能。
  - linux需要将一个页面从内存中换出时，首先查询交换缓存，如果缓存中的文件没有被修改过，则直接丢弃而不用回写外存交换文件。这样大大节省了许多不必要的磁盘操作。
- 硬件缓存

  联想寄存器

实践任务

- 以内核模块的方式，遍历mem_map数组，获得物理内存的使用情况（即free命令的输出）

```
root@kernel:~# free
              total        used        free      shared  buff/cache   available
Mem:        1024480       30996      913276        6840       80208      962800
Swap:             0           0           0
```

- 以内核模块的方式读取task_struct->mm->mmap，遍历指定进程的虚拟地址空间内存布局

```
root@kernel:~# cat /proc/1984/maps
00489000-0048a000 r--p 00000000 00:22 4981595    /tmp/share/chapter4/ShowMemoryLayout/test
0048a000-0048b000 r-xp 00001000 00:22 4981595    /tmp/share/chapter4/ShowMemoryLayout/test
0048b000-0048c000 r--p 00002000 00:22 4981595    /tmp/share/chapter4/ShowMemoryLayout/test
0048c000-0048d000 r--p 00002000 00:22 4981595    /tmp/share/chapter4/ShowMemoryLayout/test
0048d000-0048e000 rw-p 00003000 00:22 4981595    /tmp/share/chapter4/ShowMemoryLayout/test
```