第16章 存储管理

实验目的

- 了解Linux物理主存管理机制
- 了解Linux进程虚存管理机制
- 初步掌握Linux缺页异常的处理过程

主要内容

- 背景知识
  - x86的分段机制
    - ✓ I386内存寻址的硬件支持
    - ✓ 硬件及Linux的分段机制
    - ✓ 硬件及Linux的分页机制
  - 物理存储管理
  - 进程虚拟存储管理
- 实验内容
  - 统计系统和单个进程的缺页次数

i386中的段寄存器

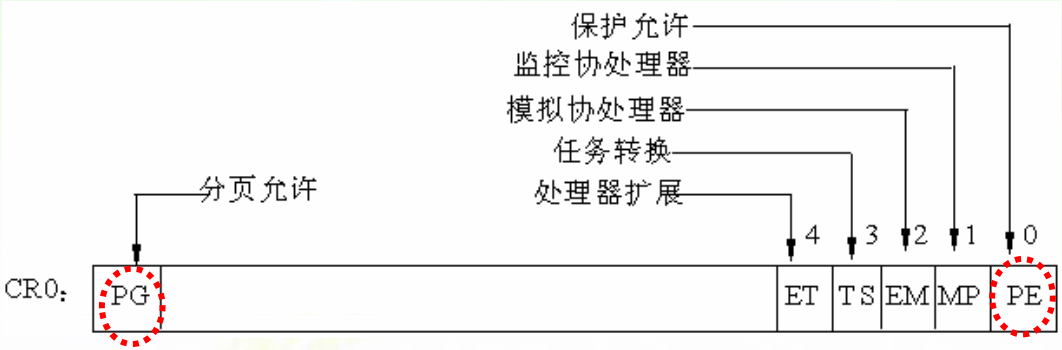
- 共6个段寄存器，每个寄存器16位
  - CS：代码段寄存器，指向存放程序指令的段
    - ✓cs寄存器包含一个两位的域，指明CPU的当前特权级CPL，值为0代表最高优先级，值为3代表最低优先级
  - SS：堆栈段寄存器，指向存放当前堆栈的段
  - DS：数据段寄存器，指向存放数据的段
  - ES、FS及GS：附加数据段寄存器
- 在保护模式下，16位的寄存器无法存放32位的段基地址，段寄存器中存放的不是某个段的基地址，而是某个段的选择子（Selector）
  - 段基地址存放在段描述符表中

i386中的状态和控制寄存器

- 由标志寄存器**EFLAGS**、指令指针**EIP**和**4**个控制寄存器组成

CR0控制寄存器

- 机器状态字
  - PE(Protedted Enable)：用于启动保护模式
    - PE=1，保护模式启动
    - PE=0，实模式下运行
  - PG(Paging Enable)：分页允许位
    - 表示芯片上的分页部件是否允许工作



PG	PE	方式
0	0	实模式，8080操作
0	1	保护模式，但不允许分页
1	0	出错
1	1	允许分页的保护模式

其他控制寄存器

- **CR1：未定义的控制寄存器**
- **CR2：页故障线性地址寄存器**
  - 保存最后一次出现页故障的全32位线性地址
- **CR3：页目录基址寄存器**
  - 保存页目录表的物理地址，页目录表总是放在以4K字节为单位的存储器边界上
  - 地址的低12位总为0，不起作用，即使写上内容，也不会被使用

i386中的系统地址寄存器

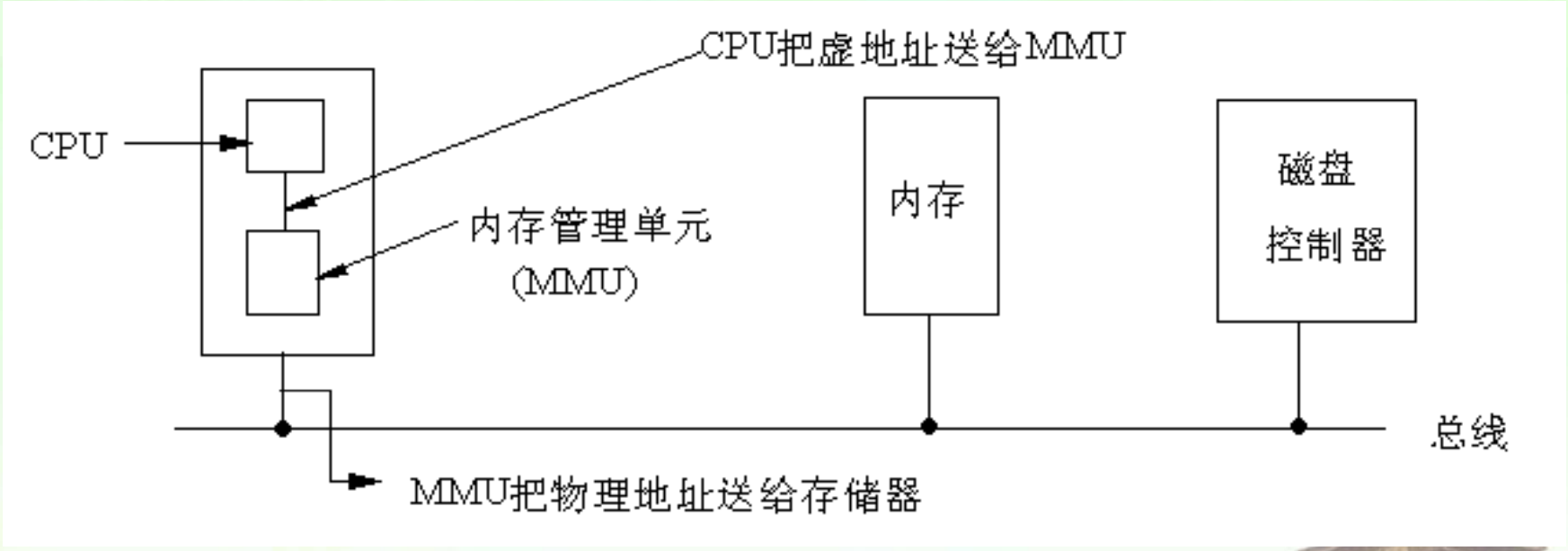
- 全局描述符表寄存器GDTR
  - 48位寄存器，保存GDT的32位基地址和16位GDT的界限
- 中断描述符表寄存器IDTR
  - 48位寄存器，保存IDT的32位基地址和16位IDT的界限
- 局部描述符表寄存器LDTR
  - 16位寄存器，保存局部描述符表LDT段的选择子
- 任务状态寄存器TSR
  - 16位寄存器，保存任务状态段TSS段的16位选择子

主要内容

- 背景知识
  - x86的分段机制
    - ✓I386内存寻址的硬件支持
    - ✓硬件及Linux的分段机制
    - ✓硬件及Linux的分页机制
  - 物理存储管理
  - 进程虚拟存储管理
- 实验内容
  - 统计系统和单个进程的缺页次数

内存寻址的一般过程

❖虚实转换



内存地址

❖Intel x86处理器的地址形式

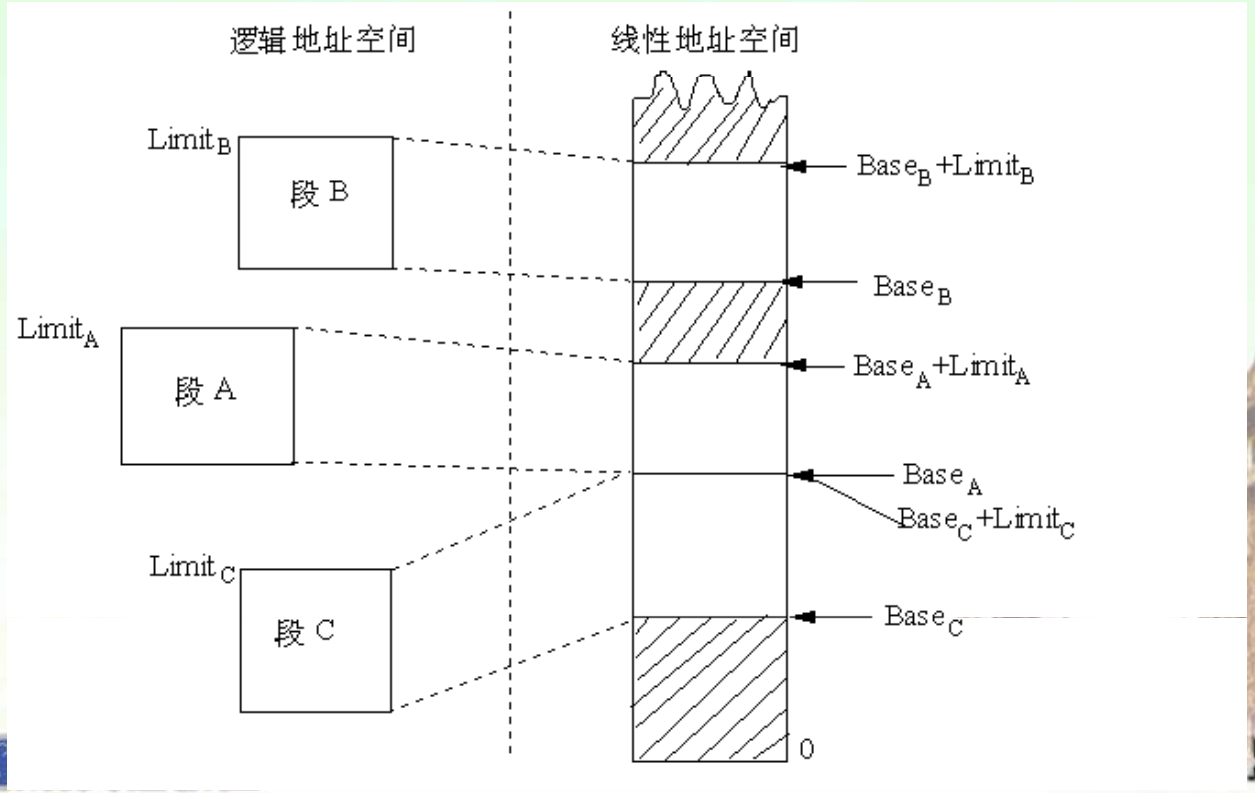
- 逻辑地址：每个逻辑地址由一个段(segment)和偏移量(offset)组成
- 线性地址：32位无符号整数，可以表示4G的地址空间
- 物理地址：用于芯片级内存单元寻址，与从CPU的地址引脚发送到内存总线上的电信号相对应

❖地址转换过程

- 内存控制单元（MMU）通过分段单元（segmentation unit）将逻辑地址转换成线性地址
- 分页单元（paging unit）将线性地址转换成一个物理地址

段机制

- 是形成逻辑地址到线性地址转换的基础
  - 基地址(Base Address): 在线性地址空间中段的起始地址
  - 界限(Limit): 逻辑地址中段内的最大偏移量
  - 段的属性(Attribute): 表示段的特性

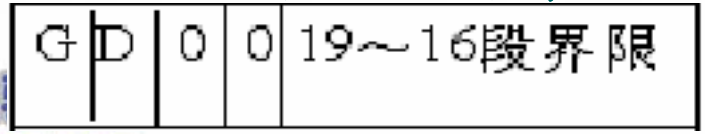


段描述符

- 每个段由一个8个字节的段描述符来表示
  - 指出段的32位基地址(base字段)和20位段界限(limit字段)
  - 第6个字节
    - ✓ G位：段界限粒度位，只对段界限有效
      - 当G=0时，表示段格式以字节长度
      - 当G=1时，表示段格式以4K字节为单位
    - ✓ D位：特殊位

段描述符—D位说明

- 描述代码段：决定指令地址及操作数的默认大小
  - D=1：指令使用32位地址及32位或8位操作数
  - D=0：指令使用16位地址及16位或8位操作数
- 向下扩展数据段：决定段的上部边界
  - D=1：段的上部界限为4G
  - D=0：段的上部界限为64K
- SS寄存器寻址的段（通常是堆栈段）：决定隐式的堆栈访问指令(如PUSH和POP指令)使用何种堆栈指针寄存器
  - D=1：使用32位堆栈指针寄存器ESP
  - D=0：使用16位堆栈指针寄存器SP



段描述符表

- 定义i386所有段的情况，所有描述符表本身都占据一个字节为8的倍数的存储器空间
- 分类
  - 全局描述符表GDT (Global Description Table)
    - ✓ 除任务门、中断门和陷阱门描述符外，包含着系统中所有任务共用段的描述符
    - ✓ 存放在RAM中，使用一个专门的寄存器GDTR指示GDT表在RAM中的位置（物理起始地址）
  - 局部描述符表LDT（Local Description Table）
    - ✓ 包含与给定任务有关的描述符
    - ✓ 存放在RAM中，使用LDTR来指示当前的LDT表
- X86的段描述符设置
  - 通常只定义一个GDT
  - 每个进程除存放在GDT中的段之外，还可以根据需要创建附加的LDT

Intel x86中的段描述符分类

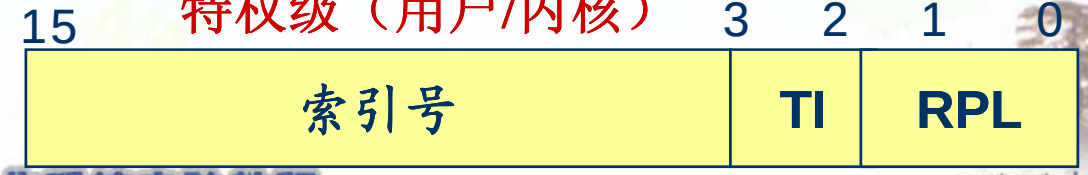
- 数据段描述符（DSD）
  - 描述各种用户数据段和堆栈段，可以放在GDT或LDT中
- 代码段描述符（CSD）
  - 描述一个用户代码段，可以放在GDT或LDT中
- 任务状态段描述符（TSSD）
  - 描述一个任务的状态段，用于保存处理器寄存器的内容，只出现在GTD中
- 局部描述符表描述符（LDTD）
  - 描述一个LDT段，只出现在GDT中
- 系统段描述符（SSD）

Intel微处理器的地址转换方式

- 实模式（20位）
  - 16位段寄存器只记录段基址的高16位，因此段基址必须4位对齐（末4位为0）
  - 不采用虚拟地址空间，直接采用物理地址空间
  - 物理地址=段寄存器值*16+段内偏移
- 保护模式（32位）
  - 16位段寄存器无法直接记录段的信息，因此需要与全局描述符表GDT配合使用
  - GDT中记录了每个段的信息（段描述符），段寄存器只需记录段在GDT中的序号
  - 线性地址=段基地址+段内偏移
    - ✓ 其中，段基地址是根据段寄存器所指明的GDT中的段描述符中的信息得到
  - 物理地址：根据页表对线性地址进行转换而得到

硬件中的分段

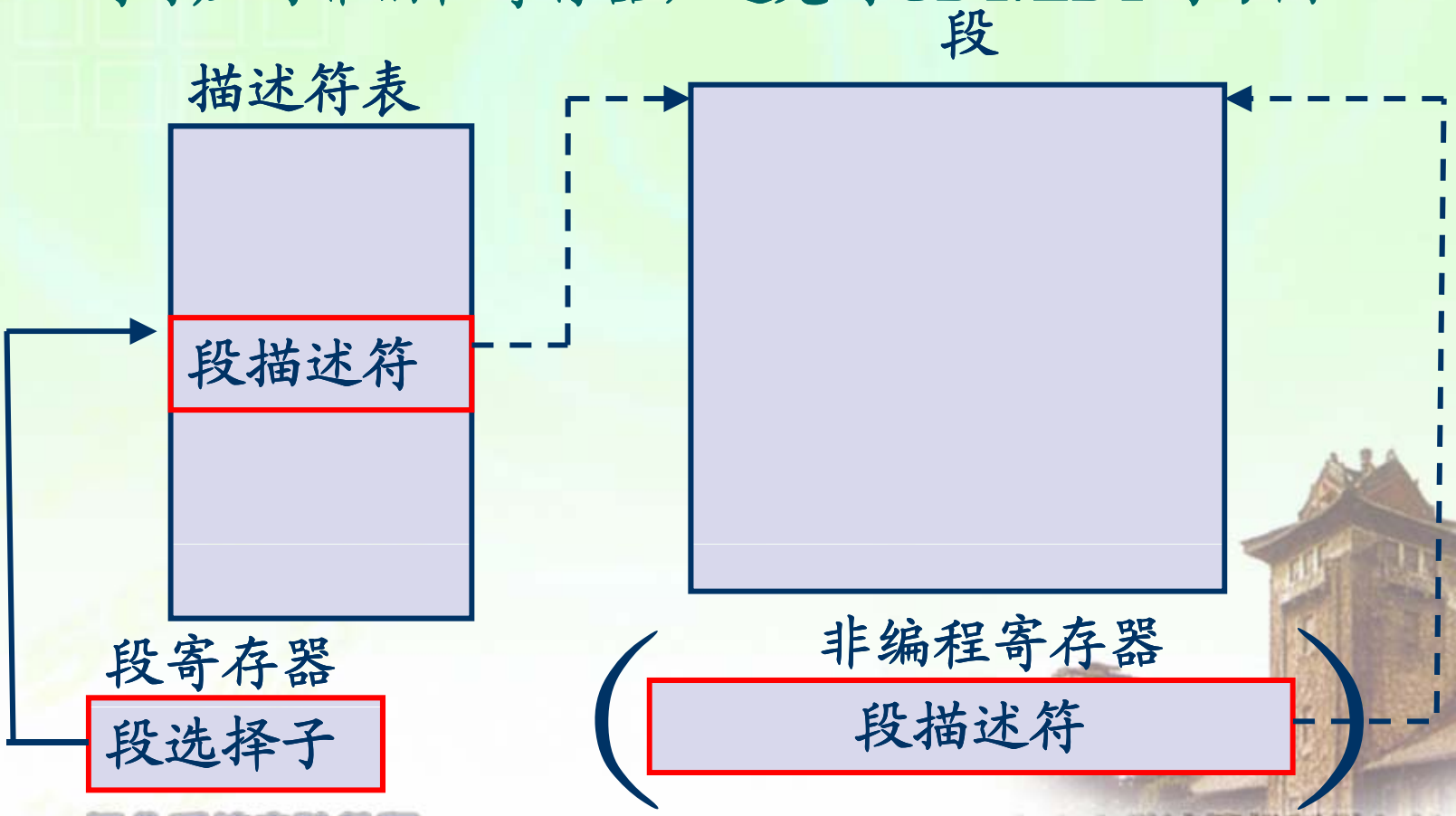
- I386体系结构采用基于段选择子的分段机制
  - 逻辑地址=段：段内偏移
    - 段标志符：16位长，称为段选择子
    - 段内偏移：32位长
  - 使用16位段寄存器来指明当前所使用的段
    - 索引号
      - 13位，指定GDT表中的相应的段描述符
    - TI(Table Indicator)
      - 1位，指明段描述符在GDT（TI=0）或LDT（TI=1）中
    - RPL(Request Privilege Level)
      - 2位，当相应段选择符装入到cs寄存器中时，指明CPU的当前特权级（用户/内核）



段选择子格式

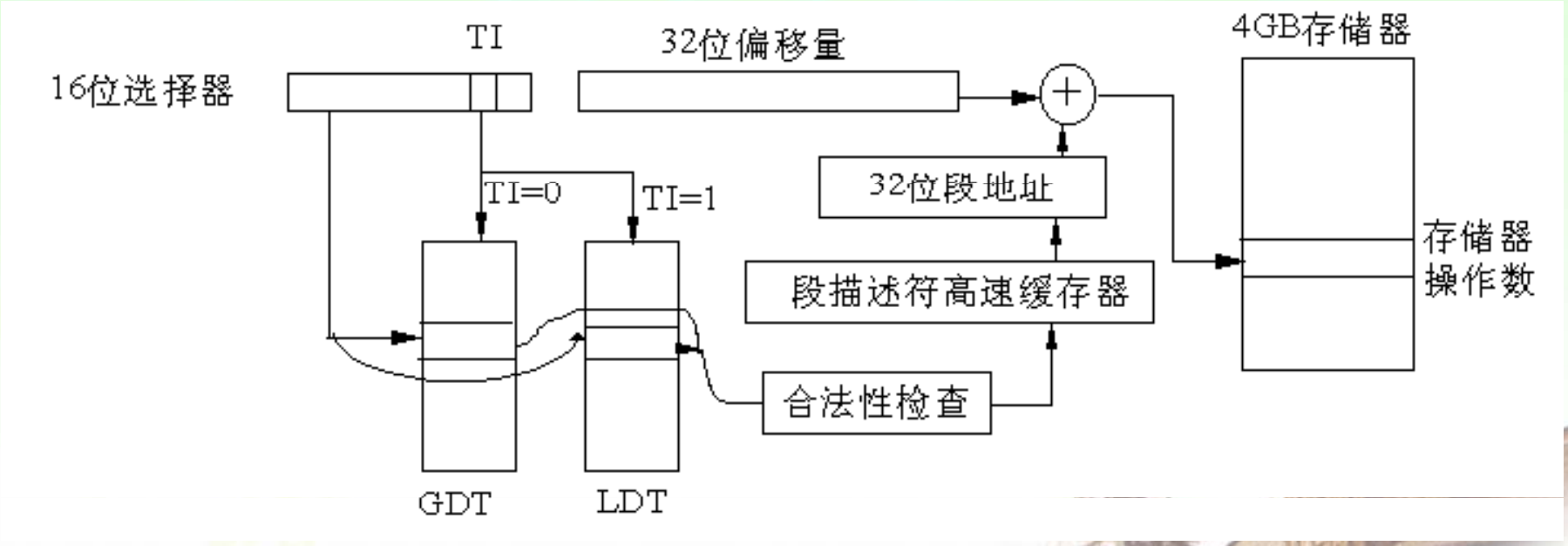
段描述符的快速访问

- 16位段寄存器与GDT/LDT配合，对相应段寻址
  - 在段选择子被装入段寄存器时，将相应的段描述符装入到对应的非编程寄存器，避免对GDT/LDT的访问



逻辑地址到线性地址的转换

- 检查TI确定段描述符位置
- 从段选择子的index字段计算段描述符的地址
- 将逻辑地址的偏移量与段描述符base字段的值相加，得到线性地址



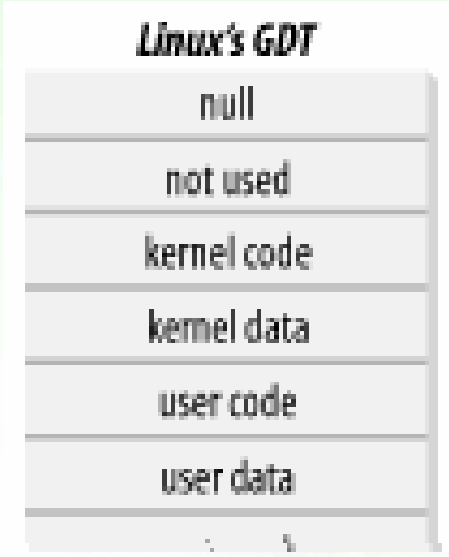
Linux中的段

❖ Linux中的段设计思想

- 段和页的同时存在在一定程度上有点多余，因为两者都可以划分进程的物理空间
  - ✓ 分段可以给每个进程分配不同的线性地址空间
  - ✓ 分页可以把同一线性地址空间映射到不同的物理空间
- 所有的进程希望使用同样的**0-4G**的逻辑空间
- 这样程序员不必考虑进程地址的问题，也让内核的内存管理变得简单一些

❖ Linux的段结构

```
#define __KERNEL_CS 0x10
#define __KERNEL_DS 0x18

#define __USER_CS   0x23
#define __USER_DS   0x2B
```



Linux中的段

❖ Linux中的段设计思想

- 段和页的同时存在在一定程度上有点多余，因为两者都可以划分进程的物理空间
  - ✓ 分段可以给每个进程分配不同的线性地址空间
  - ✓ 分页可以把同一线性地址空间映射到不同的物理空间
- 所有的进程希望使用同样的**0-4G**的逻辑空间
- 这样程序员不必考虑进程地址的问题，也让内核的内存管理变得简单一些

❖ Linux的段结构

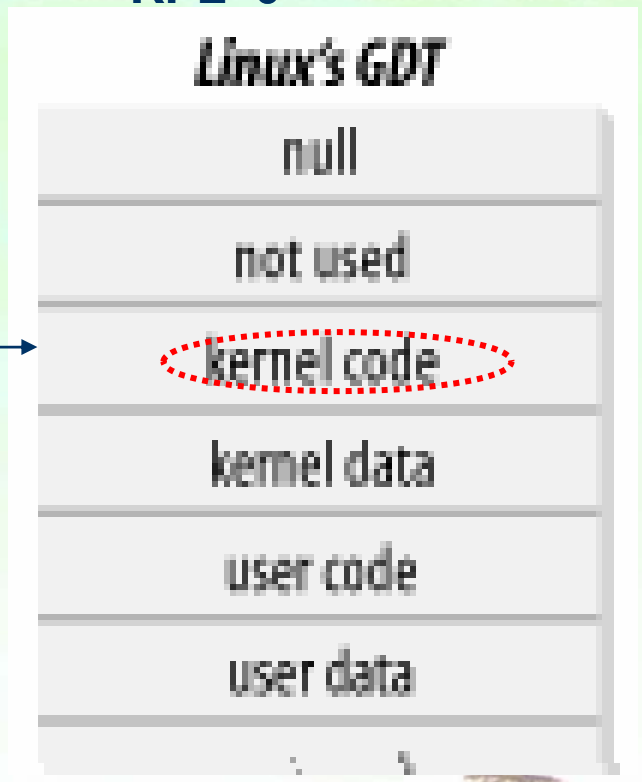
```
#define __KERNEL_CS 0x10
#define __KERNEL_DS 0x18

#define __USER_CS   0x23
#define __USER_DS   0x2B
```

__KERNEL_CS

❖ 0x10=0000 0000 0001 0000b

Index=2

RPL=0 ← 特权级



__KERNEL_CS

❖ 内核代码段，在GDT中相应的段描述符各个域有如下值

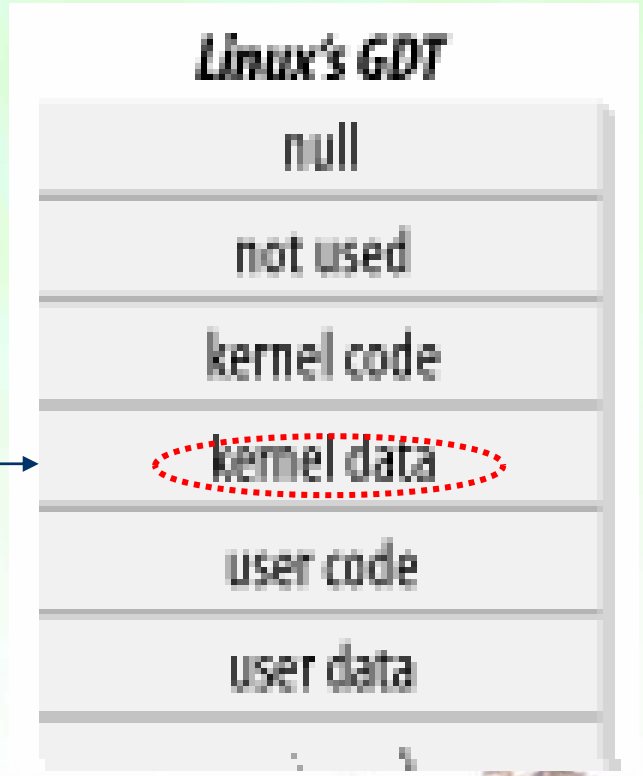
4GB
- Base = 0x00000000
- Limit = 0xfffff
- G (granularity flag) = 1, for segment size expressed in pages
- S (system flag) = 1, for normal code or data segment
- Type = 0xa, for code segment that can be read and executed
- DPL (Descriptor Privilege Level) = 0, for Kernel Mode
- D/B (32-bit address flag) = 1, for 32-bit offset addresses

__KERNEL_DS

- 0x18=0000 0000 0001 1000b

Index=3

RPL=0 ← 特权级



__KERNEL_DS

- 内核数据段，在GDT中相应的段描述符各个域有如下值

4GB
- Base = 0x00000000
- Limit = 0xfffff
- G (granularity flag) = 1, for segment size expressed in pages
- S (system flag) = 1, for normal code or data segment
- Type = 2, for data segment that can be read and written
- DPL (Descriptor Privilege Level) = 0, for Kernel Mode
- D/B (32-bit address flag) = 1, for 32-bit offset addresses

__USER_CS

❖用户代码段，用户态下所有进程共享

- Base = 0x00000000
- Limit = 0xfffff
- G (granularity flag) = 1, for segment size expressed in pages
- S (system flag) = 1, for normal code or data segment
- Type = 0xa, for code segment that can be read and executed
- DPL (Descriptor Privilege Level) = 3, for User Mode
- D/B (32-bit address flag) = 1, for 32-bit offset addresses

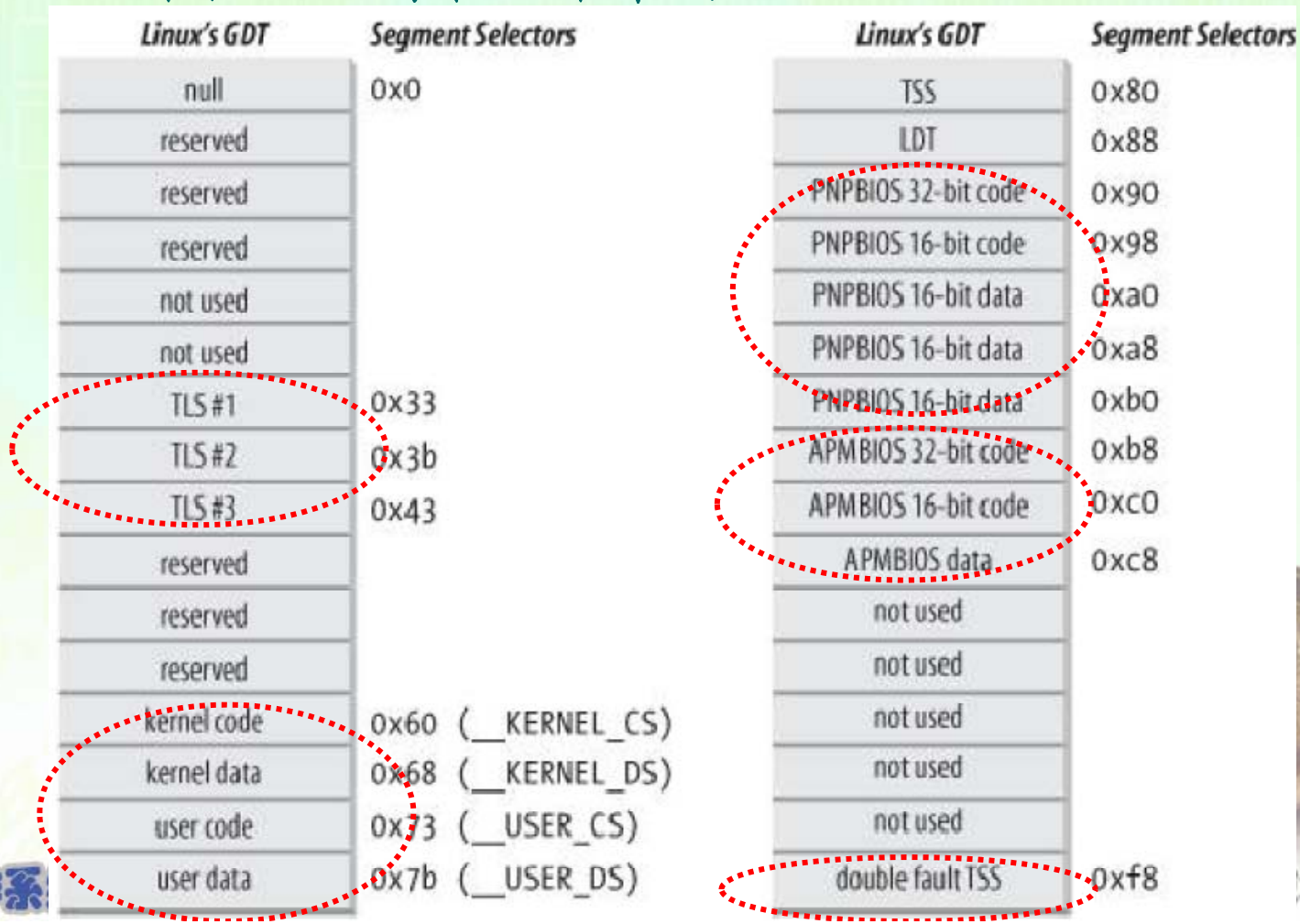
__USER_DS

❖用户数据段，用户态下所有进程共享

- Base = 0x00000000
- Limit = 0xfffff
- G (granularity flag) = 1, for segment size expressed in pages
- S (system flag) = 1, for normal code or data segment
- Type = 2, for data segment that can be read and written
- DPL (Descriptor Privilege Level) = 3, for User Mode
- D/B (32-bit address flag) = 1, for 32-bit offset addresses

Linux下的GDT

- 每个CPU对应一个GDT
  - 包括18个段描述符和14个未用项



Linux下的LDT

- 大多数用户态下的Linux程序不使用LDT
- 内核定义一个缺省的LDT让大多数进程共享
  - 存放在default_ldt数组中
  - 包括5项
  - 内核仅有效使用了其中的两项
    - ✓iBCS执行文件的调用门
    - ✓Solaris/X86的可执行文件的调用门

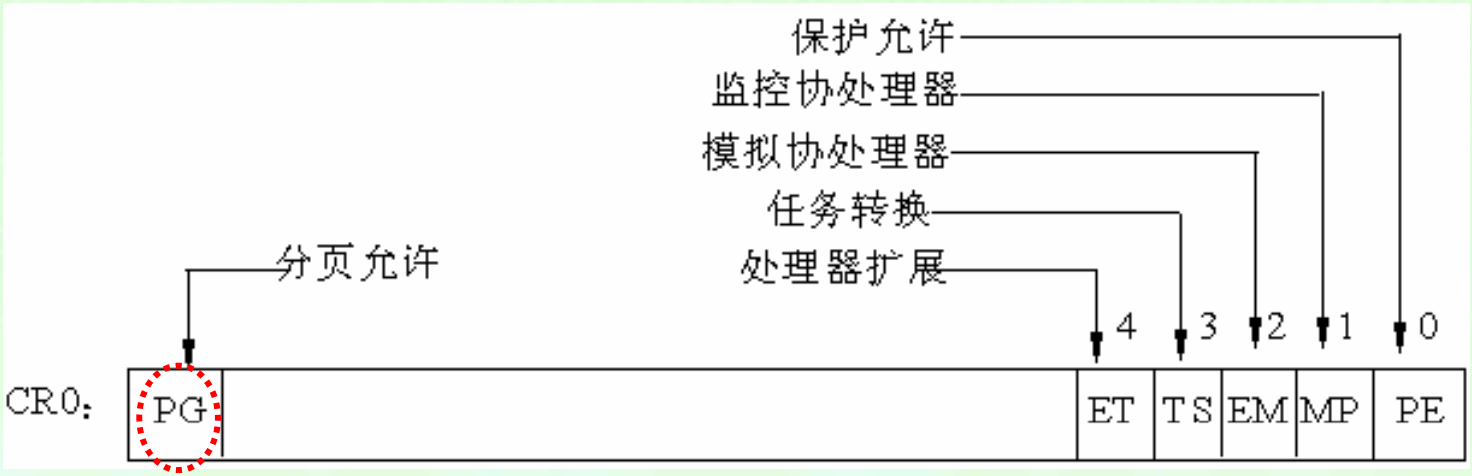
主要内容

- 背景知识
  - x86的分段机制
    - ✓I386内存寻址的硬件支持
    - ✓硬件及Linux的分段机制
    - ✓硬件及Linux的分页机制
  - 物理存储管理
  - 进程虚拟存储管理
- 实验内容
  - 统计系统和单个进程的缺页次数

硬件中的分页

- 分页单元（paging unit）
  - 功能：线性地址→物理地址
  - 关键任务
    - 把所请求的访问类型与线性地址的访问权限比较，若此次内存访问无效，则产生一个缺页异常
  - 页(page)
    - 将线性地址分成以固定长度为单位的组
  - 页表(page table)
    - 将线性地址映射到物理地址的数据结构
    - 存放在内存中，并在启用分页单元以前由内核对之进行初始化
  - 页框(page frame)
    - 把所有RAM划分成以固定长度为单位的组
    - 每个页框可以包含一页，即页框长度等于页的长度
  - Intel处理器中，通过设置CR0寄存器的PG标志位来启用分页单元
    - 当PG=0时，线性地址被解释成物理地址

CR0控制寄存器

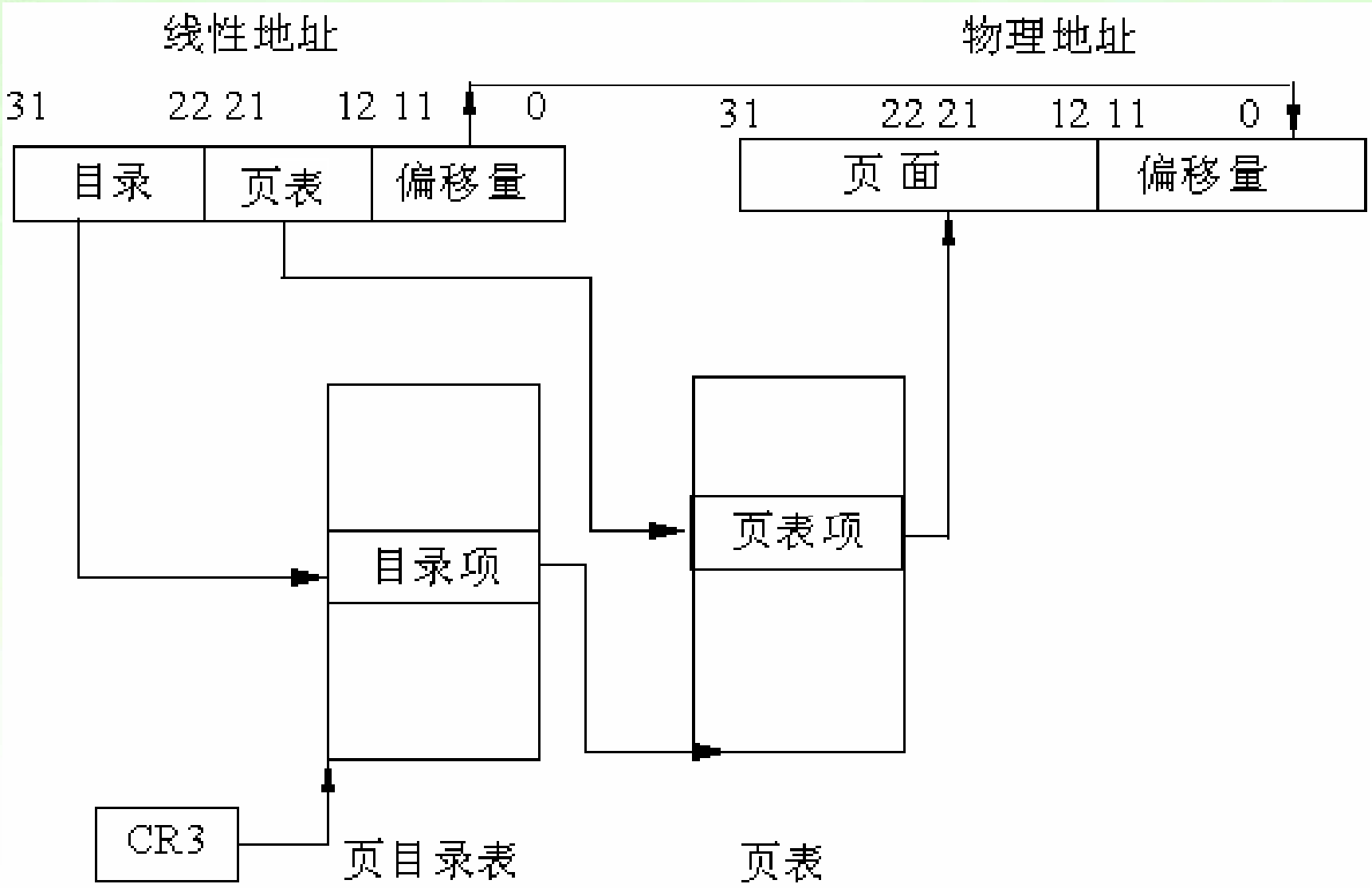


PG	PE	方式
0	0	实模式，8080操作
0	1	保护模式，但不允许分页
1	0	出错
1	1	允许分页的保护模式

常规分页

- 从i386起，Intel处理器的分页单元处理4KB的页
- 32位的线性地址被分成3个域
  - 目录(directory)：最高的10位，决定页目录项（指向适当的页表）
  - 页表(table)：中间的10位，决定页表项（指向所在页框的物理地址）
  - 偏移量(offset)：最低的12位，决定页框内的相对位置
- 线性地址的转换（二级模式）
  - 分两步完成，每一步都基于一种转换表
    - ✓ 第一种称为页目录表(page directory)
    - ✓ 第二种称为页表(page table)
  - 正在使用的页目录表的物理地址存放在CPU的CR3寄存器中

Intel 80x86处理器的分页



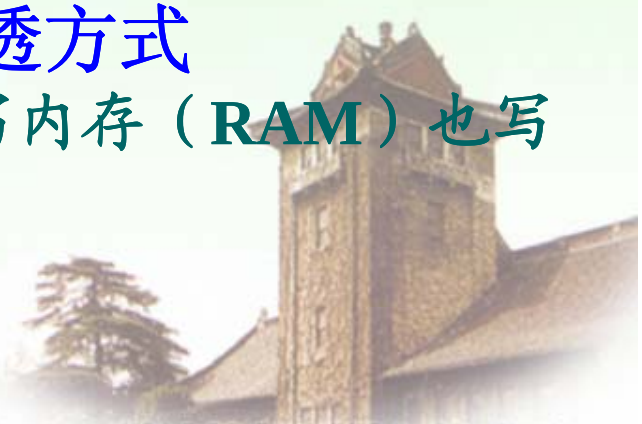
页目录表项结构

- 每个页目录项4字节，最多1024项
  - 为什么在线性地址中页表及页目录只要用10位表示？
- 页表地址的低12位总为0，所以用高20位指出32位页表地址即可

	7	6	5	4	3	2	1	0
7~0位	PSE	0	A	PCD	PWT	U/S	R/W	P
15~8位	3~0位页表地址				OS专用			0
23~16位	11~4位页表地址							
31~24位	19~12位页表地址							

页目录表项属性说明

- 存在位P（第0位）
  - 如果P=1，表示页表地址指向的该页在内存中
  - 如果P=0，表示不在内存中，分页单元就把这个线性地址存放在处理器的CR2寄存器中，并产生一个14号异常（缺页异常）
- 读/写位R/W（第1位）及用户/管理员U/S位（第2位）
  - 这两位为页目录项提供硬件保护
    - ✓特权级为3的进程要访问页面时需通过页保护检查
    - ✓特权级为0的进程可绕过页保护
- PWT（第3位）位：是否采用写透方式
  - 为1表示采用写透方式，该方式既写内存（RAM）也写高速缓存



页目录表项属性说明

- **PCD（第4位）位：是否启用高速缓存**
  - 该位为1表示启用高速缓存
- **访问位（第5位）**
  - 当对页目录项进行访问时，A位为1
- **Page Size标志（第7位）：只适用于页目录项**
  - 如果置为1，页目录项指的是4MB的页面（扩展分页）
- **操作系统专用（第9~11位）：Linux未使用**

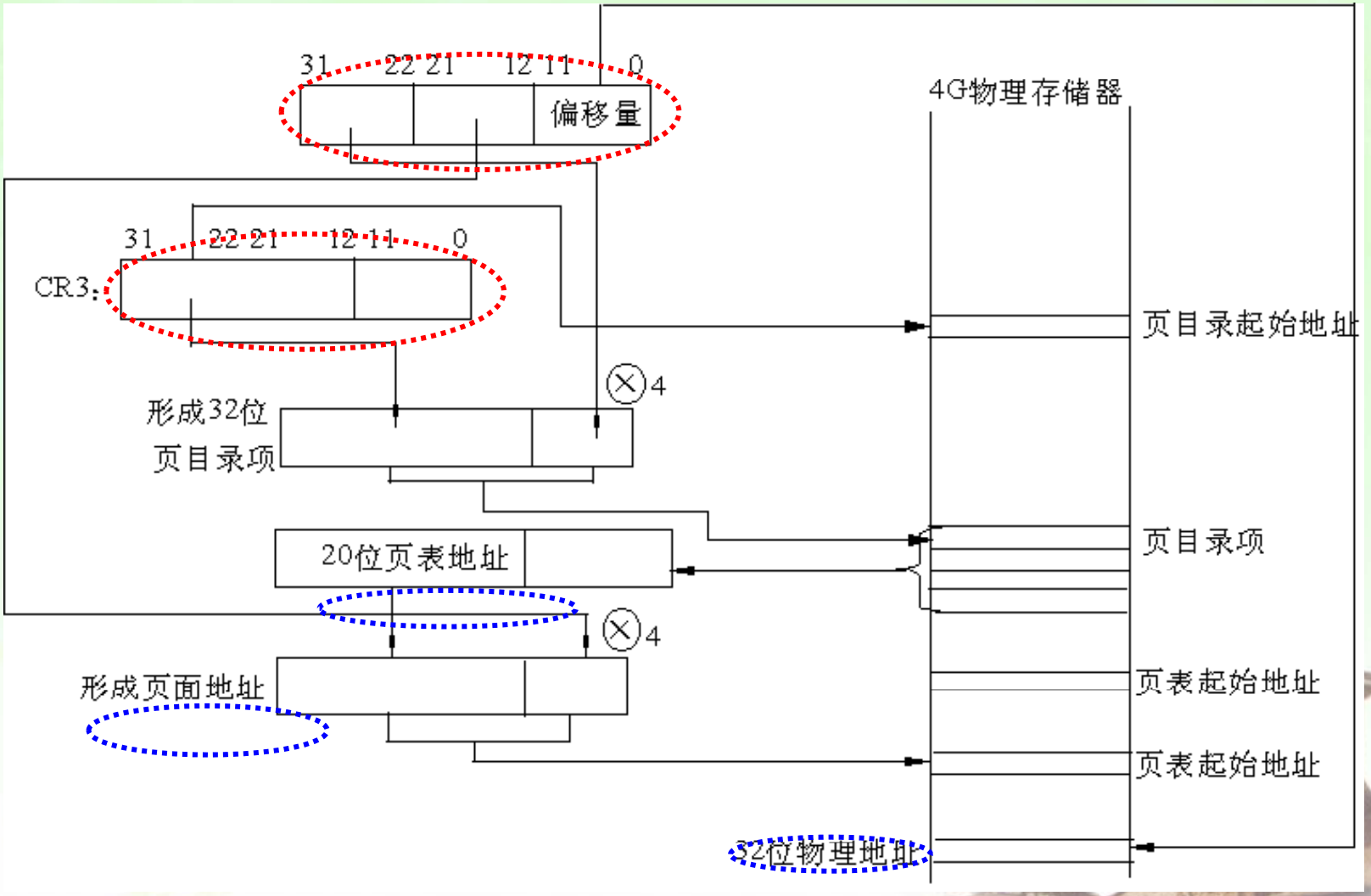
页表项结构

- 结构与页目录表项相同
  - 包含页面的起始地址和有关该页面的信息
    - ✓ 页面的起始地址也是4K的整数倍
    - ✓ 页面的低12位也留作它用
  - 除第6位外，第0～5位及9~11位的用途和页目录项一样
    - ✓ 第6位是页面项独有的，当对涉及的页面进行写操作时，D位被置1

二级模式线性地址转换的特点

- 减少每个进程页表所需RAM的数量
  - 一级页表缺陷
    - 使用一级页表需要2^{20}个表项（每项4字节，需要4MB）
  - 二级模式特点
    - 每个活动进程必须有一个页目录，但不必立即为进程的所有页表分配RAM
    - 只为进程实际使用的那些虚拟内存区请求页表来减少内存需求
    - 页目录和页表都可以多达1024项（2^{10}）
    - 一个页目录的寻址能力1024*1024*4096=2^{32}

线性地址到物理地址的转换

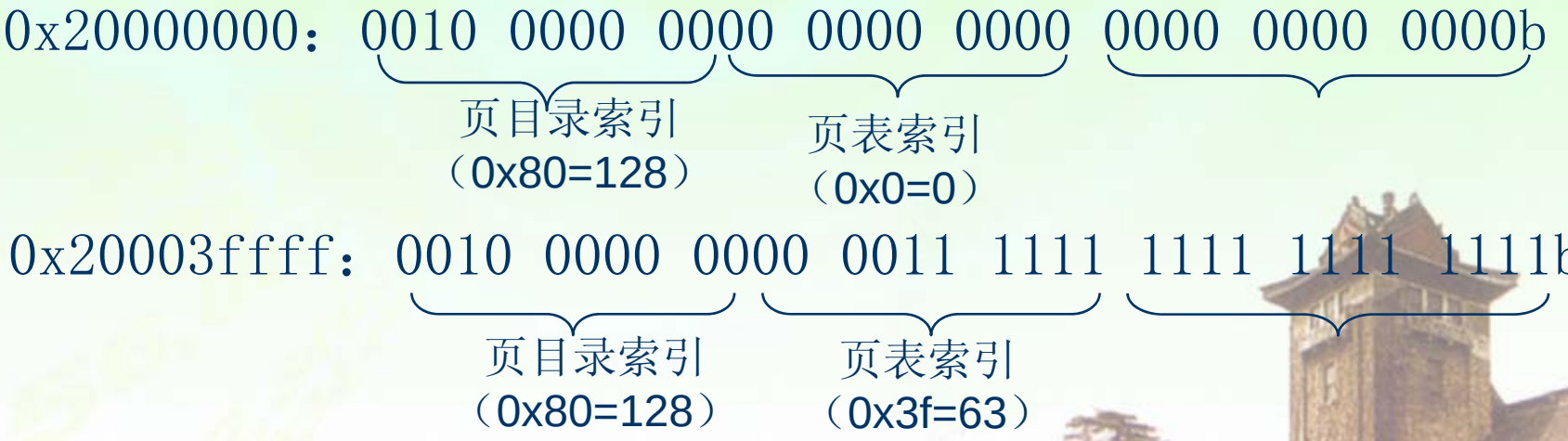


线性地址到物理地址的转换步骤

- 第一步：形成页表地址
  - CR3包含着页目录的起始地址，用32位线性地址的最高10位A31~A22作为页目录的页目录项的索引，将它乘以4(每项4个字节)，与CR3中的页目录的起始地址相加，形成相应页表地址
- 第二步：形成页面地址
  - 从指定的地址中取出32位页目录项，它的低12位为0，这32位是页表的起始地址。用32位线性地址中的A21~A12位作为页表中的页面的索引，将它乘以4，与页表的起始地址相加，形成32位页面地址
- 第三步：形成32位物理地址
  - 将A11~A0作为相对于页面地址的偏移量，与32位页面地址相加，得到物理地址

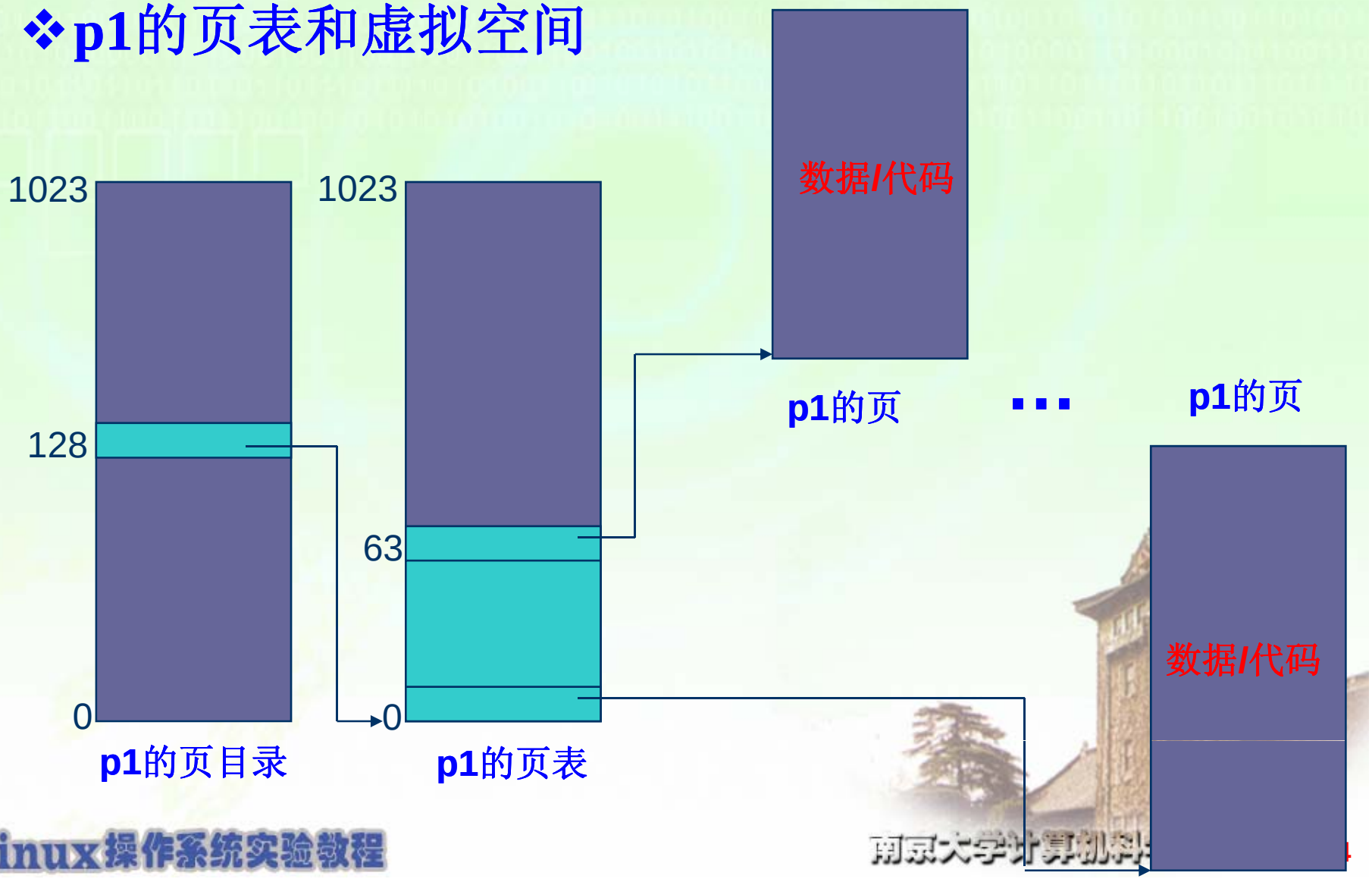
常规分页举例

- 假设内核给一个正在运行的进程p1分配的线性地址空间是0x20000000到0x2003ffff（256KB）
  - 该段空间大小为0x40000，即0x40个页（64页）
  - 有效的线性地址范围为



常规分页举例

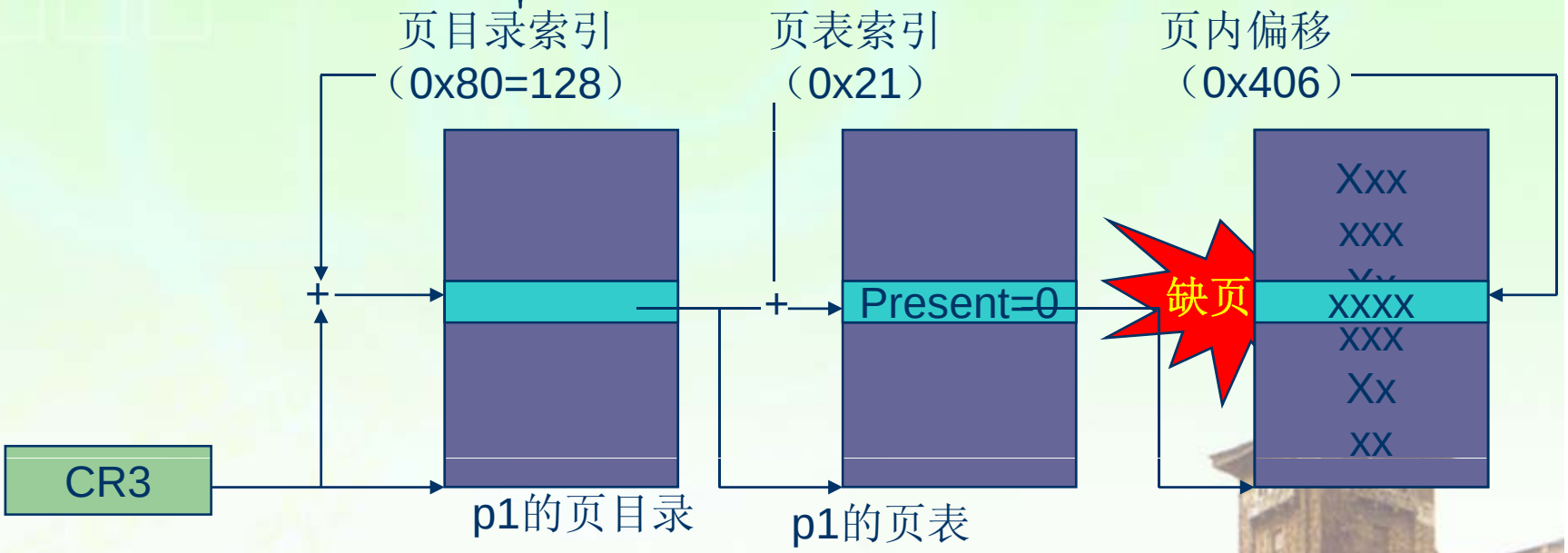
❖p1的页表和虚拟空间



分页举例

- 假设进程需要读取0x20021406中的字节
- 分页单元将该地址划分为3个部分

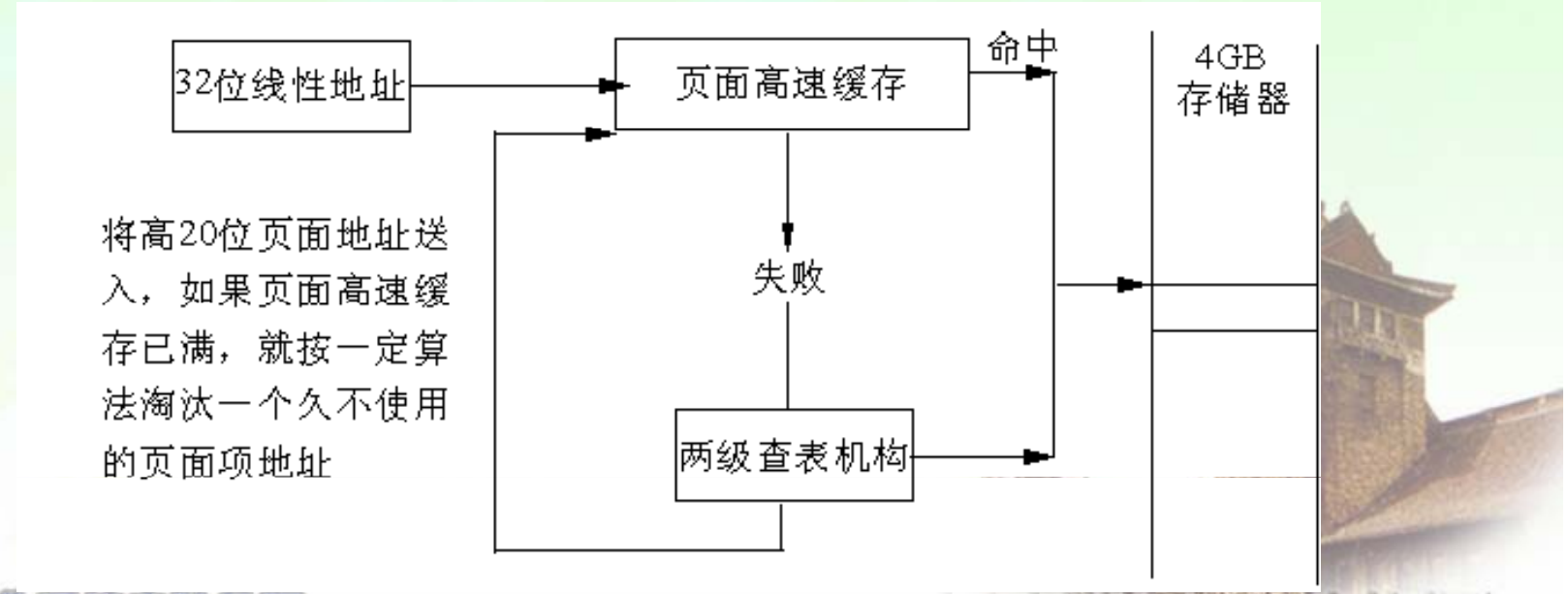
0x20021406=0010 0000 0000 0010 0001 0100 0000 0110b



- 当进程无论何时试图访问0x20000000到0x2003ffff范围之外的线性地址时，都将产生一个保护错误

转换后援缓冲器TLB

- 用于加速线性地址的转换
  - 当一个线性地址第一次被访问时，访问RAM中的页表计算出物理地址，并将物理地址存放在TLB表项
  - 随后访问时首先查找TLB
  - 当CR3控制寄存器被修改时，硬件自动使本地TLB的所有项失效



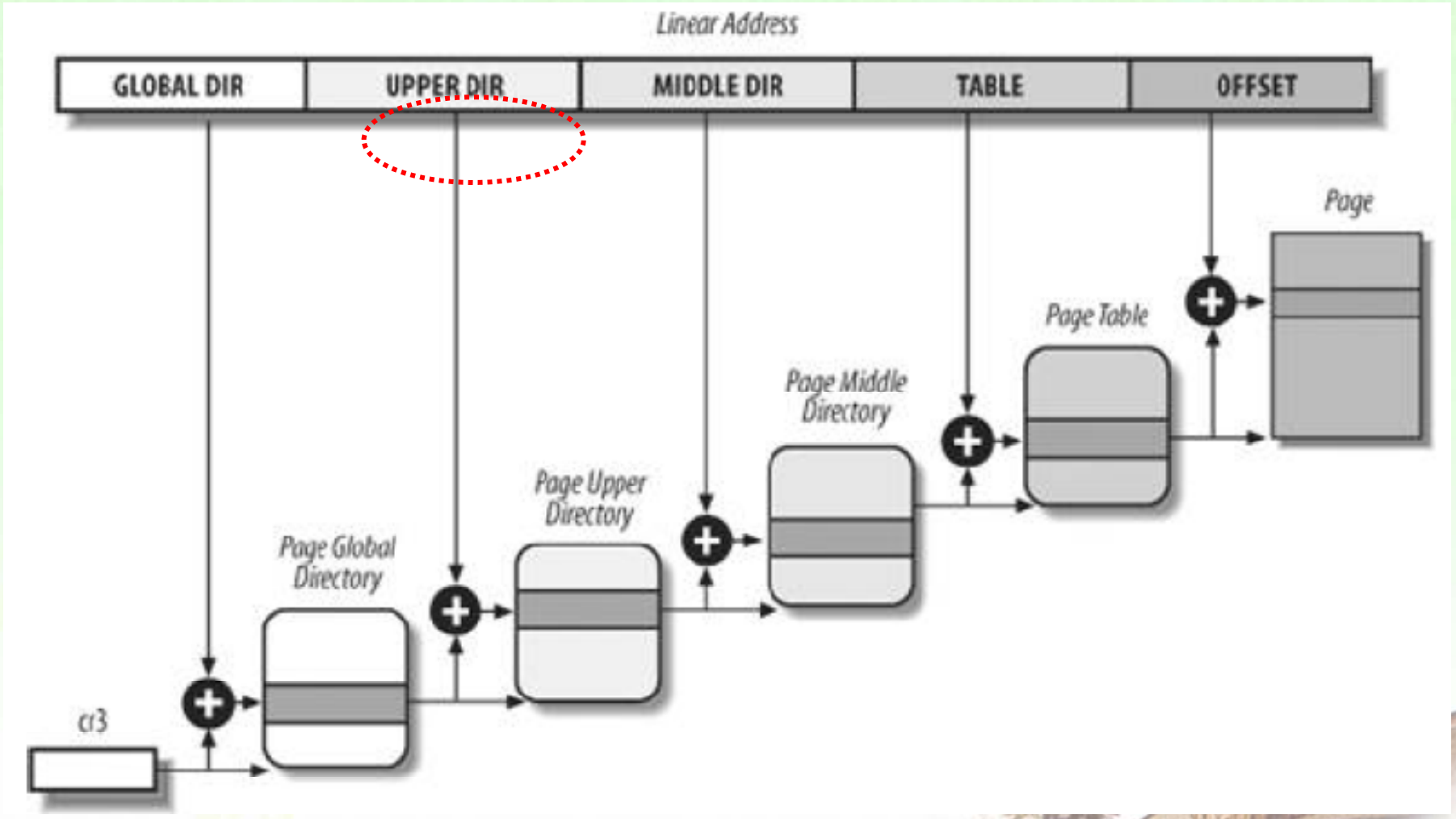
Linux进程的分页机制

- **Linux的分段机制**
  - 所有进程使用相同的段寄存器值及线性地址空间（0～4G）
- **Linux对进程的处理很大程度上依赖于分页**
  - 实际上，由硬件提供的**MMU**将线性地址自动转换为物理地址使得一下设计目标变得可行
    - ✓ 给每个进程分配不同的物理地址空间，该机制可确保对寻址错误提供有效保护
    - ✓ 区别页(即一组数据)和页框(实际的物理空间)之间的不同。这是虚拟存储器机制的基本因素
  - 每个进程都有它自己的页全局目录和自己的页表集合
    - ✓ 当进程切换发生时，Linux把CR3寄存器的值保存在跟进程相关的一个数据结构中，然后用另外一个进程相应的值填充CR3寄存器
    - ✓ 当新进程恢复在CPU上执行时，分页单元将使用一组与新进程对应的页表

Linux的分页模型

- 采用同时适用32位和64位系统的普通分页模型
  - Linux 2.6.10以前版本使用三级分页模型
  - Linux 2.6.10以后版本使用四级分页模型
    - 页全局目录PGD(Page Global Directory)
    - 页上级目录PUD(Page Upper Directory)
    - 页中间目录PMD(Page Middle Directory)
    - 页表PT(Page Table)
  - 没有启用物理地址扩展的32位系统，使用两级页表
    - 将“页上级目录”位和“页中间目录”位全设置为0
  - 启用物理地址扩展的32位系统使用三级页表
    - 全局目录对应X86的PDPT，取消“页上级目录”
    - 页中间目录对应X86的页目录
    - 页表对应X86的页表
  - 64位系统使用三级或四级页表，取决于硬件对线性地址的划分

Linux 2.6的四级分页模式



主要内容

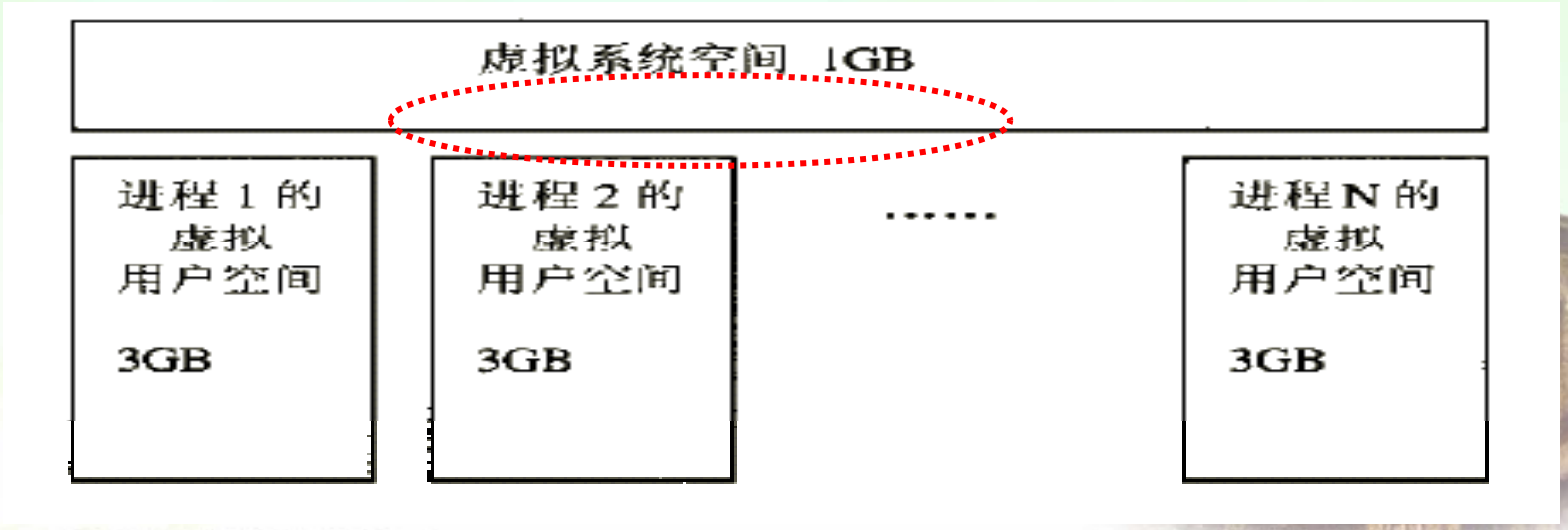
- 背景知识
  - x86的分段机制
  - 物理存储管理
    - ✓ 页框管理
    - ✓ 内存区管理
    - ✓ 非连续存储区管理
  - 进程虚拟存储管理
  - slab分配器
- 实验内容
  - 统计系统和单个进程的缺页次数

内存寻址的主要工作

- Linux如何有效地利用x86的分段和分页机制把逻辑地址转换为物理地址
- RAM的某些部分永久地分配给内核，用以存放内核代码以及静态数据
- RAM其余部分的管理？
  - 线性地址空间的管理
  - 进程地址空间的管理

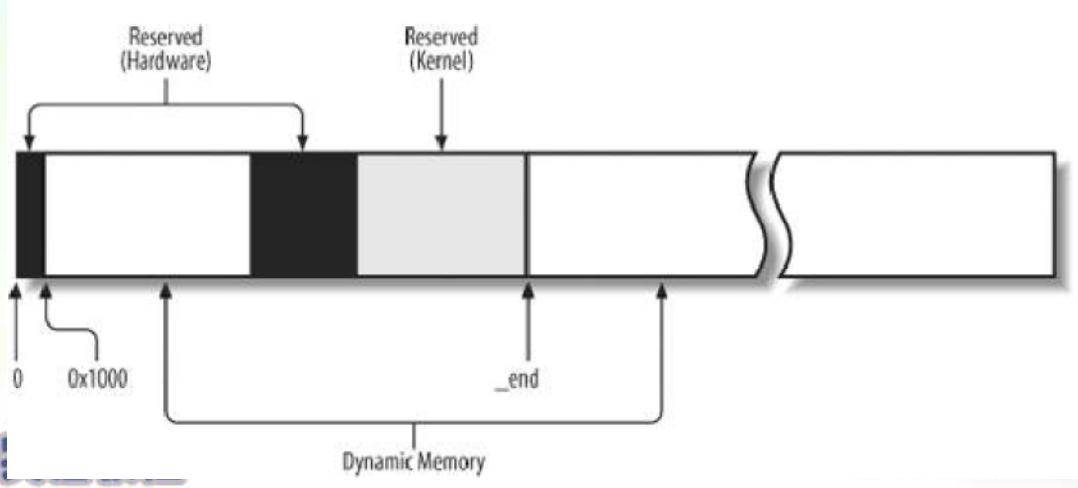
Linux的虚拟空间

- 每个进程拥有3G用户空间
- 内核占用最高1G作为系统空间
  - 系统空间由所有进程共享
- 地址转换
  - 通过页表把虚存空间的一个地址转换为物理空间中的实际地址



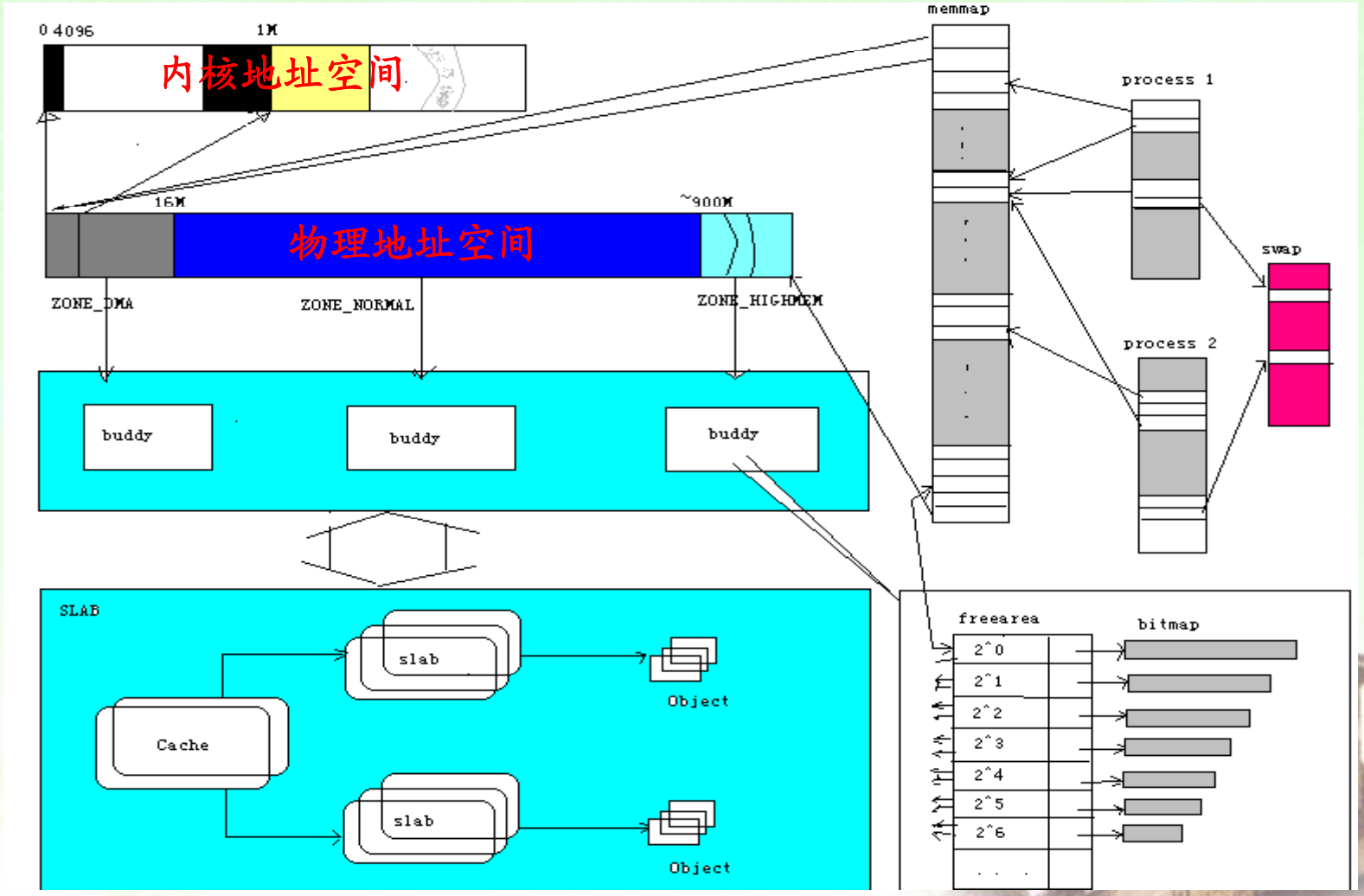
Linux的虚拟空间管理

- 内存管理
  - 动态管理内核线性虚拟空间
    - ✓ 物理地址空间到虚拟地址空间的映射
    - ✓ 属于稀缺资源，页框及内存区的分配与优化
      - 按需分配，不需要时释放
- 进程地址空间
  - 管理进程虚拟地址空间
    - ✓ 编程地址空间与进程虚拟地址空间的映射
    - ✓ 进程虚拟地址空间与物理地址空间的映射



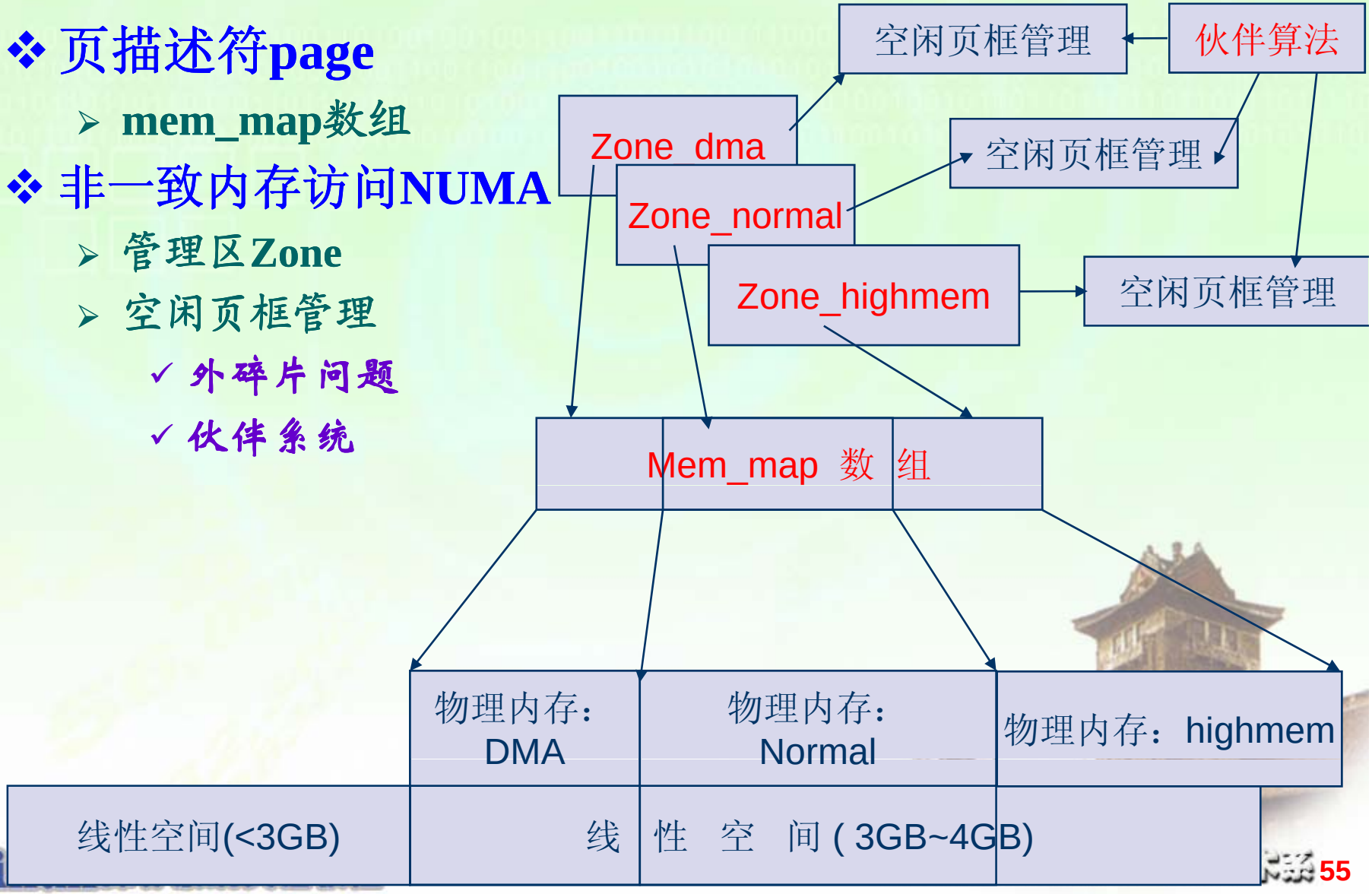
内存管理的核心工作

进程



页框管理主要内容

- 页描述符page
  - mem_map数组
- 非一致内存访问NUMA
  - 管理区Zone
  - 空闲页框管理
    - ✓ 外碎片问题
    - ✓ 伙伴系统



页框设计

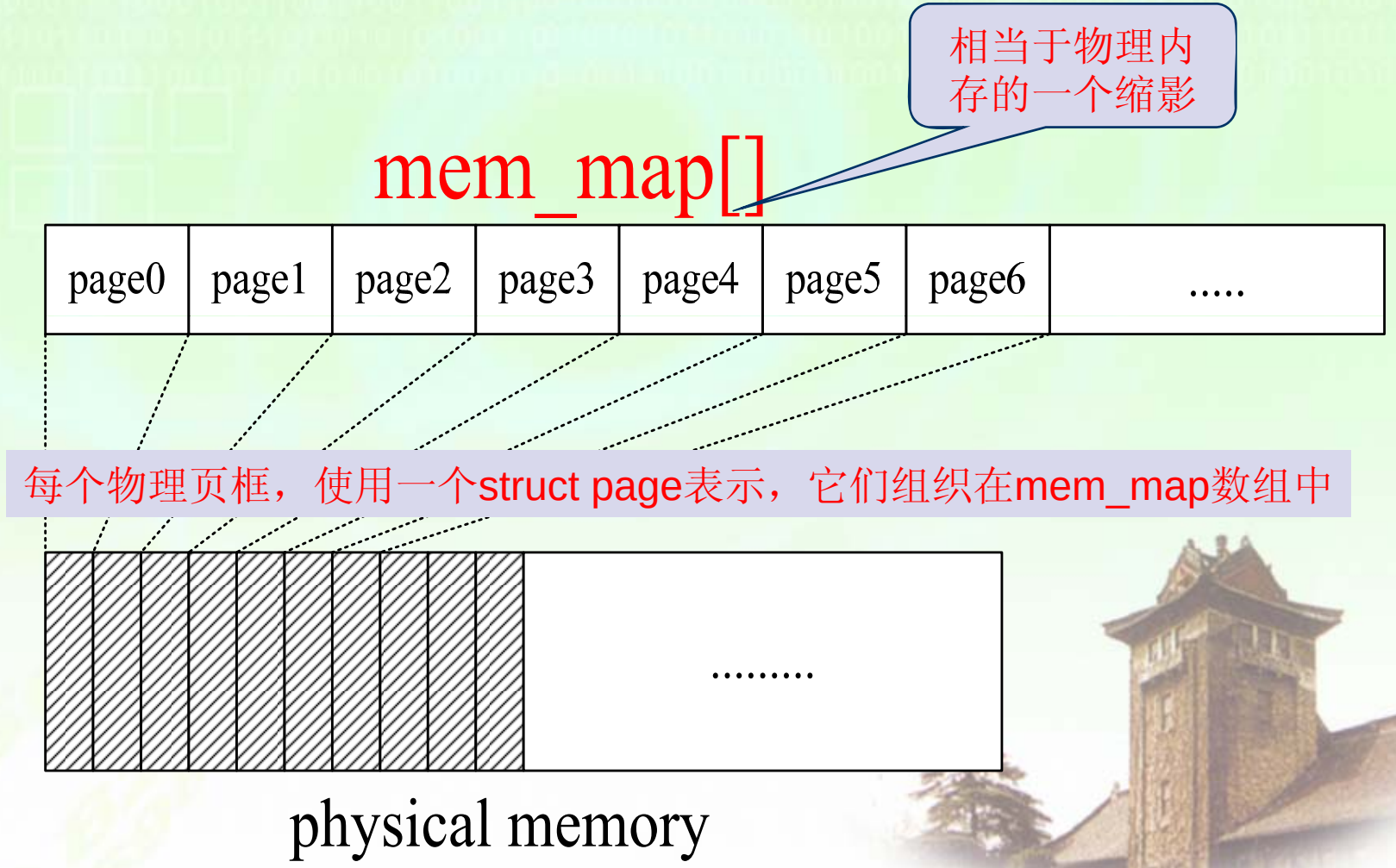
- 采用页作为内存管理的基本单位
- 标准页框大小为**4KB**
  - **4KB是大多数磁盘块大小的倍数**
    - ✓ **传输效率高**
    - ✓ **管理方便**

```
/* PAGE_SHIFT determines the page size */
#define PAGE_SHIFT   12
#define PAGE_SIZE    (1UL << PAGE_SHIFT)
#define PAGE_MASK    (~(PAGE_SIZE-1))
```

页描述符

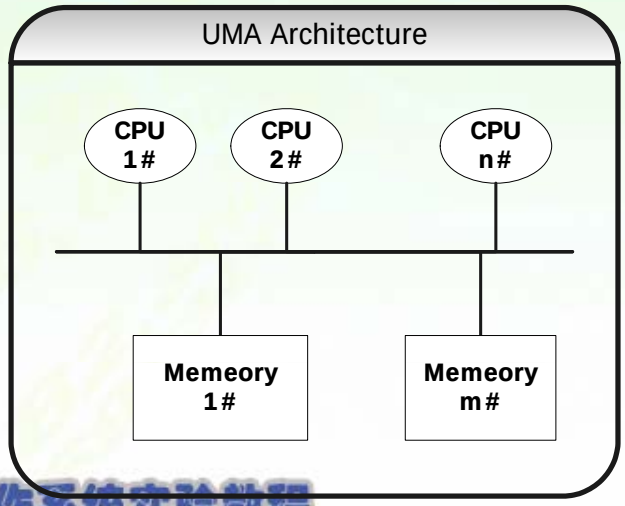
- 内核使用页描述符跟踪管理物理内存
  - 每个物理页框都用一个页描述符表示
  - 结构类型： **struct page [include/linux/Mm_types.h]**
    - ✓ 每个描述符长度为32字节
  - 所有物理页框的描述符，组织在**mem_map**的数组中
    - ✓ 所需空间略小于整个RAM的1%
    - ✓ virt_to_page(addr)宏产生线性地址addr对应的页描述符地址
    - ✓ pft_to_page(pfn)宏产生与页框号pfn对应的页描述符地址

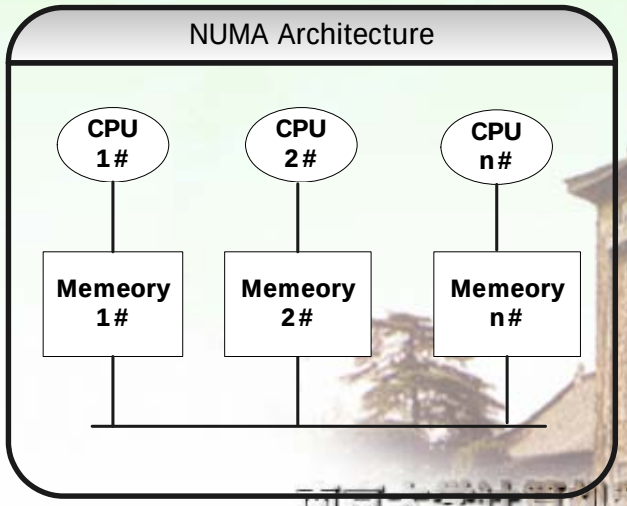
mem_map数组


相当于物理内存的一个缩影
mem_map[]
page0
page1
page2
page3
page4
page5
page6
.....
每个物理页框，使用一个struct page表示，它们组织在mem_map数组中
.....
physical memory

非一致内存访问NUMA

- 计算机内存不是一种均匀的资源，给定CPU对不同存储器单元的访问延迟可能不一样
- Linux将物理内存划分成多个节点(node)
  - 物理内存划分成若干节点（node），每个节点通常由一组CPU和本地内存组成的
  - 给定CPU对单独节点内的存储单元的访问延迟相同
  - 采用节点、管理区（zone）和页描述物理内存





节点描述符

- 结构名：**pg_data_t**
  - 所有节点的描述符存放在一个单向链表中
  - 第一个元素由**pgdat_list**变量指向，**x86**只有一个节点，其描述符存在**contig_page_data**变量中
  - 每个节点中的物理内存可分为几个管理区

```
typedefstructpglist_data{
  //zone类型描述符号数组
  struct zone_t[] node_zones;
  struct zonelist_t[] node_zonelists;
  int nr_zones; //该节点的zone个数
  struct page *node_mem_map;
  struct bootmem_data *bdata;
  //该节点的起始物理地址
  unsigned long node_start_pfn;
  unsigned long node_present_pages; /* 物理页总数*/
  /* 物理页范围的整个大小包括空洞*/
  unsigned long node_spanned_pages;
  int node_id;
  struct pglist_data *pgdat_next;
  wait_queue_head_t kswapd_wait;
  struct task_struct *kswapd;
} pg_data_t;
```

管理区zone

- 在理想的体系结构中，一个页框就是一个物理存储单元，可以用于任何事情，如
  - 存放内核数据/用户数据/缓存磁盘数据等
- 实际上存在硬件制约
  - 一些页框由于自身的物理地址的原因不能被一些任务所使用
    - ISA总线的DMA控制器只能对RAM的前16M寻址
    - 在一些具有大容量RAM的32位计算机中，CPU不能直接访问所有的物理存储器，因为线性地址空间不够
- **Linux将具有同样性质的物理内存划分成管理区(zones)**

管理区描述符

```
typedef struct zone_struct {
    /*
     * Commonly accessed fields:
     */
    spinlock_t          lock;
    unsigned long       free_pages;
    unsigned long       pages_min, pages_low, pages_high;
    int                 need_balance;

    /*
     * free areas of different sizes
     */
    free_area_t         free_area[MAX_ORDER];

    /*
     * Discontig memory support fields.
     */
    struct pglist_data  *zone_pgdat;
    struct page         *zone_mem_map;
    unsigned long       zone_start_paddr;
    unsigned long       zone_start_mapnr;

    /*
     * rarely used fields:
     */
    char                *name;
    unsigned long       size;
} ? end zone_struct ?   zone_t;
```

Zone中的空闲页数

Zone中的空闲页面管理数据结构

Zone中页框描述符数组

Zone中的第一个页框的物理地址

Zone中第一个页框描述符的下标

Zone的名字：“DMA”或“Normal”或“HighMem”

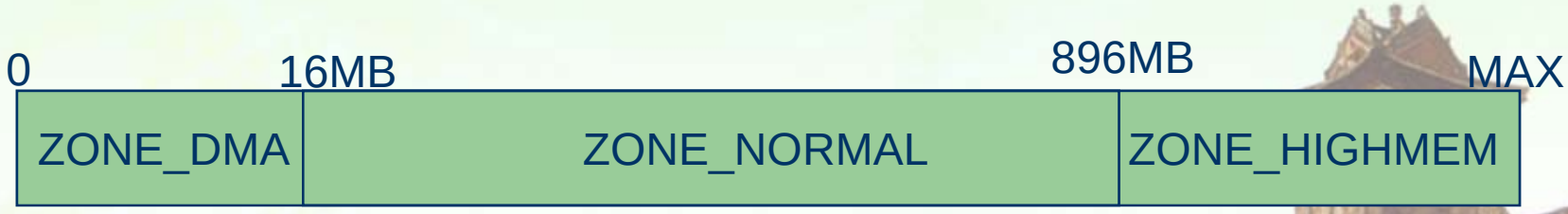
Zone中的页数

Linux在x86上的管理区

- 分成3个区
  - 每个管理区使用**struct zone_struct**表示

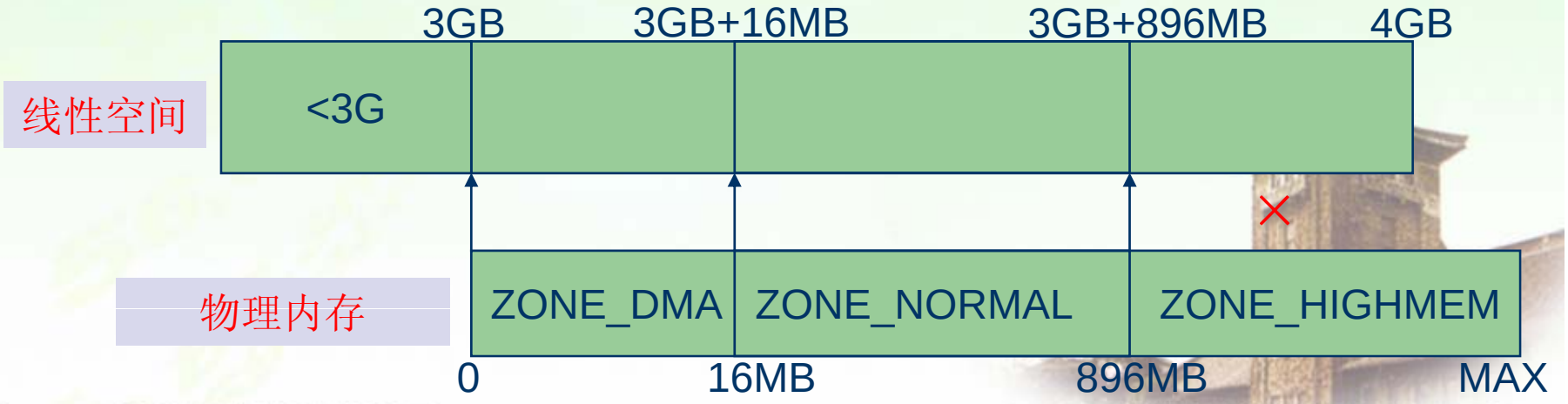
```
#define ZONE_DMA        0
#define ZONE_NORMAL     1
#define ZONE_HIGHMEM    2
#define MAX_NR_ZONES    3
```

- 0 → 低于16MB的物理页
- 1 → 高于16MB且低于896MB的物理页
- 2 → 高于896MB的物理页

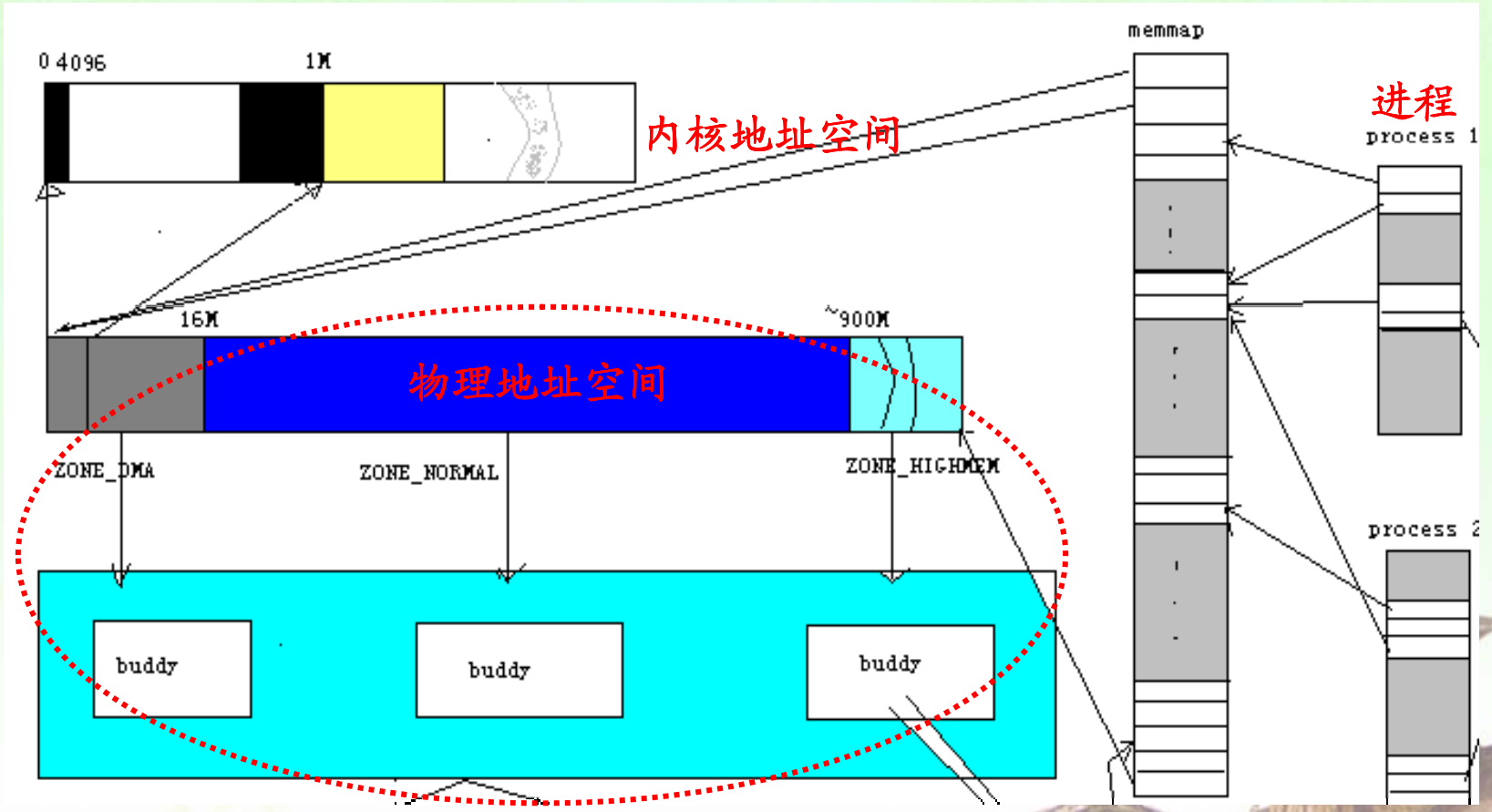


Linux在x86上的管理区说明

- **ZONE_DMA和ZONE_NORMAL区**
  - 包含存储器的常规页
  - 映射到线性地址空间的3GB以上，内核可直接访问
- **ZONE_HIGHMEN区**
  - 包含的存储器页面不能由内核直接访问

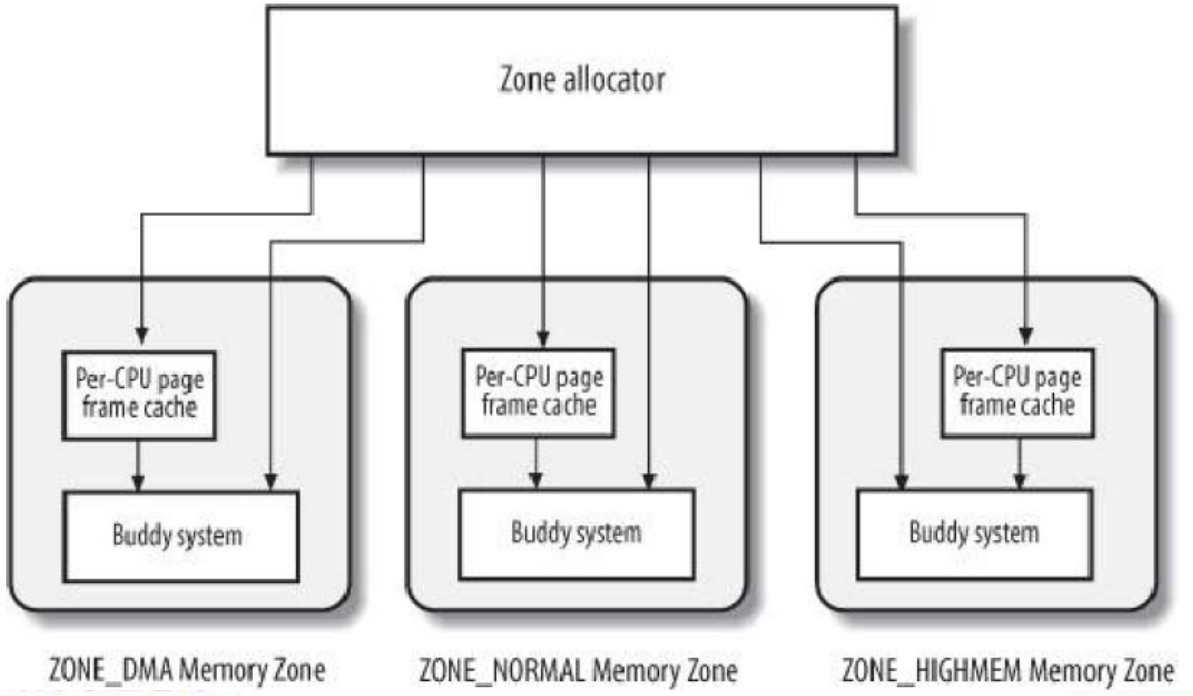


页框管理的基本结构


页描述符数组
memmap
0 4096
1M
内核地址空间
进程
process 1
16M
~900M
物理地址空间
ZONE_DMA
ZONE_NORMAL
ZONE_HIGHMEM
process 2
buddy
buddy
buddy

管理区页框分配器

- 对连续页框组的内存分配请求
  - 接受动态内存分配与释放的请求
  - 页框被命名为伙伴系统（Buddy system）
  - 每个管理区分配器有不同的伙伴系统
  - 高速页框缓存用于快速满足对单个页框的分配请求



页框管理的外碎片问题

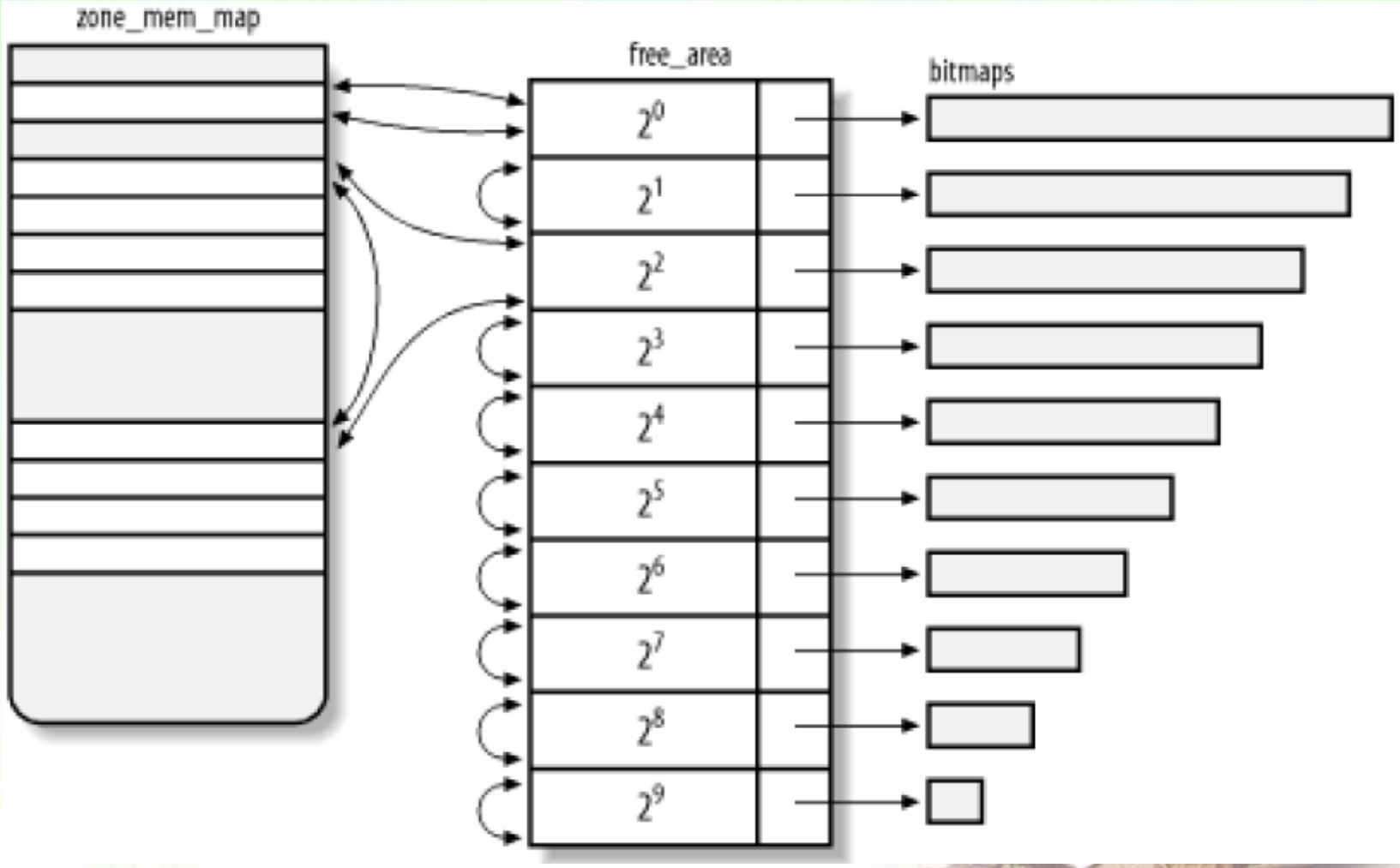
- 内核连续页框分配请求导致外碎片问题
  - 频繁的请求和释放不同大小的一组连续页框，必然导致在物理页框中分散许多小块的空闲页框
  - 这样，即使有足够的空闲页框页框满足请求，但要分配一个大块的连续页框可能就无法满足
- 可行解决途径
  - 利用MMU把一组非连续的物理空闲页框映射到连续的线性地址空间
  - 记录现存的空闲连续页框的情况，尽量避免为满足对小块的请求而把大块的空闲块进行分割
- Linux内核首选第二种方法
  - 在某些情况下，必须使用连续的页框，如DMA
  - 尽量少的修改内核页表

伙伴系统

❖基本思想

- 把所有空闲页框分组为10个块链表
  - ✓每个块链表分别包含大小为1，2，4，8，16，32，64，128，256和512个连续的页框
  - ✓每个块的第一个页框的物理地址是该块大小的整数倍
    - 例如：大小为16个页框的块，其起址是16×4KB的倍数

伙伴系统的结构示意图



管理区描述符与伙伴系统相关的数据结构

```
typedef struct zone_struct {
        /*
         * Commonly accessed fields:
         */
        spinlock_t              lock;
        unsigned long           free_pages;
        unsigned long           pages_min, pages_low, pages_high;
        int                     need_balance;

        /*
         * free areas of different sizes
         */
        free_area_t             free_area[MAX_ORDER];

        /*
         * Discontig memory support fields.
         */
        struct pglist_data      *zone_pgdat;
        struct page             *zone_mem_map;
        unsigned long           zone_start_paddr;
        unsigned long           zone_start_mapnr;

        /*
         * rarely used fields:
         */
        char                    *name;
        unsigned long           size;
} ? end zone_struct ?   zone_t;
```

Zone中的空闲页数

Zone中的空闲页面管理数据结构

基于管理区的伙伴系统

- Linux为每个zone使用独立的伙伴系统
  - Zone DMA
  - Zone Normal
  - Zone HighMem
- 每个伙伴系统使用的主要数据结构
  - 页描述符数组zone_mem_map，每个管理区都关系到mem_map的一个子集
    - ✓子集的第一个元素和元素个数分别由管理区描述符的zone_mem_map和size字段指定
  - 结构为free_area_t的空闲内存管理数组free_area
  - 与空闲内存管理数组相关的位图数组map

管理区描述符

```
typedef struct zone_struct {
        /*
         * Commonly accessed fields:
         */
        spinlock_t              lock;
        unsigned long           free_pages;
        unsigned long           pages_min, pages_low, pages_high;
        int                     need_balance;

        /*
         * free areas of different sizes
         */
        free_area_t             free_area[MAX_ORDER];

        /*
         * Discontig memory support fields.
         */
        struct pglist_data      *zone_pgdat;
        struct page             *zone_mem_map;
        unsigned long           zone_start_paddr;
        unsigned long           zone_start_mapnr;

        /*
         * rarely used fields:
         */
        char                    *name;
        unsigned long           size;
} ?  end zone_struct ?   zone_t;
```

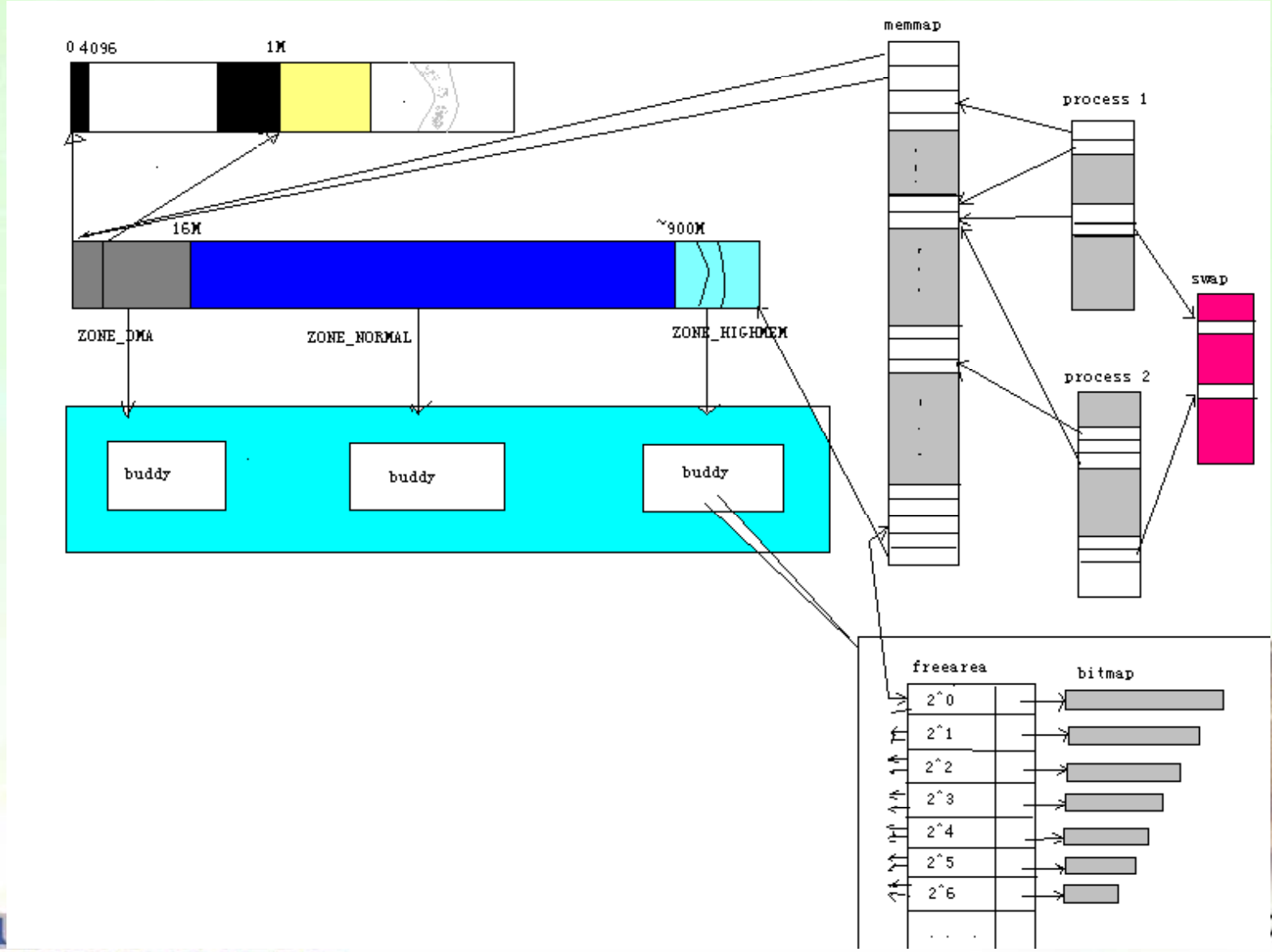
free_area_struct结构定义

```
typedef struct free_area_struct {
    struct list_head    free_list;
    unsigned long       *map;
} free_area_t;
```

所有大小为2^k个页框的块组成的双向循环链表；链表中的每个元素是对应块的第一个页框描述符

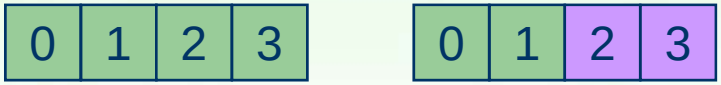
map字段指向一个位图。位图的每一位描述大小为2^k个页框的两个伙伴块的状态。

基于管理区的伙伴系统全局视图



伙伴的定义

- 伙伴的必要条件
  - 大小相同
  - 物理地址连续
  - 假定伙伴的大小为b，第一个伙伴的第一个页框的物理地址必须是2 × b × 4KB的倍数
- 事实上伙伴是通过对大块的物理内存划分获得的
  - 假如从第0个页面开始到第3个页面结束的内存

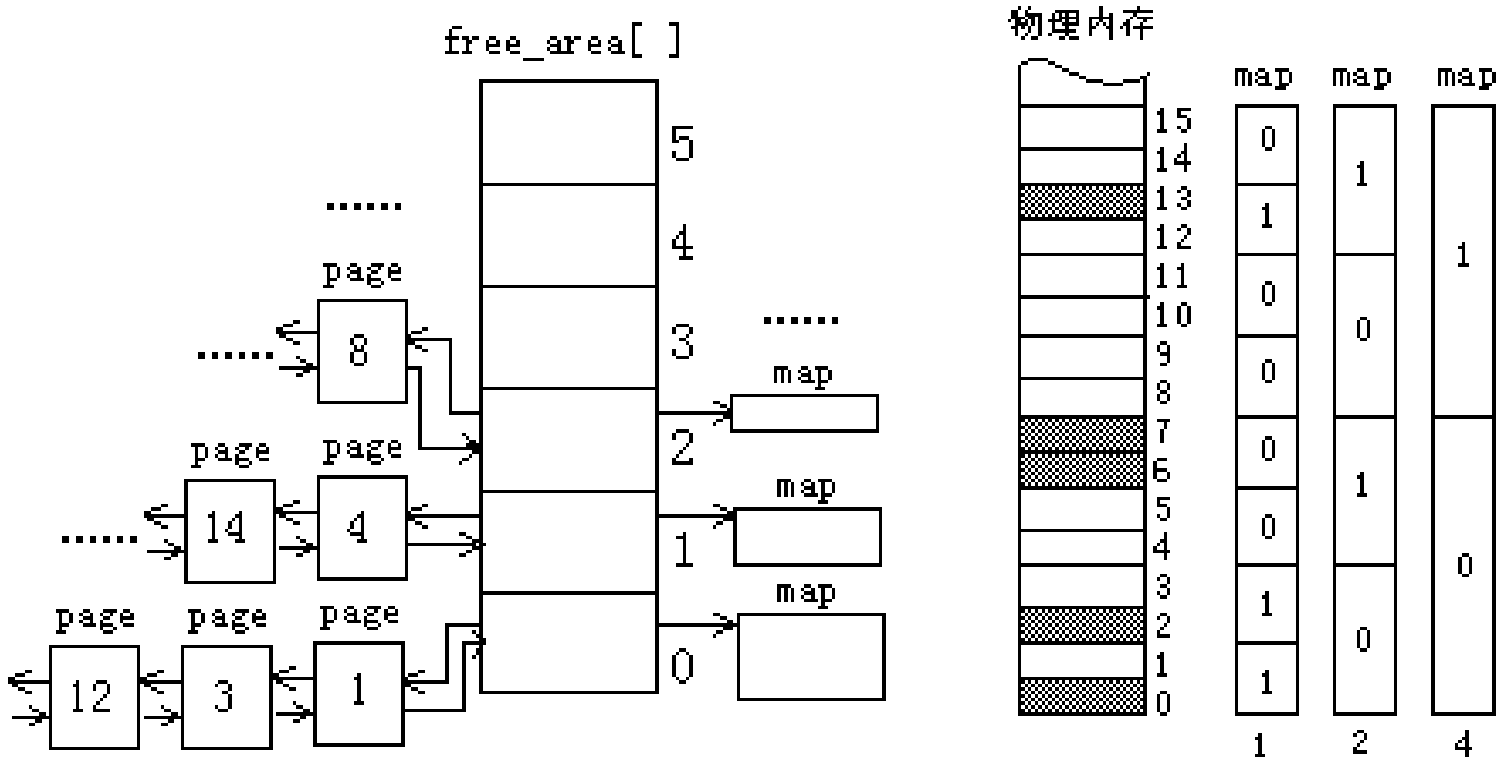


  - 每次都对半划分，那么第一次划分获得大小为2页的伙伴
  - 进一步划分，可以获得大小为1页的伙伴，例如0和1，2和3

伙伴的合并及位图含义

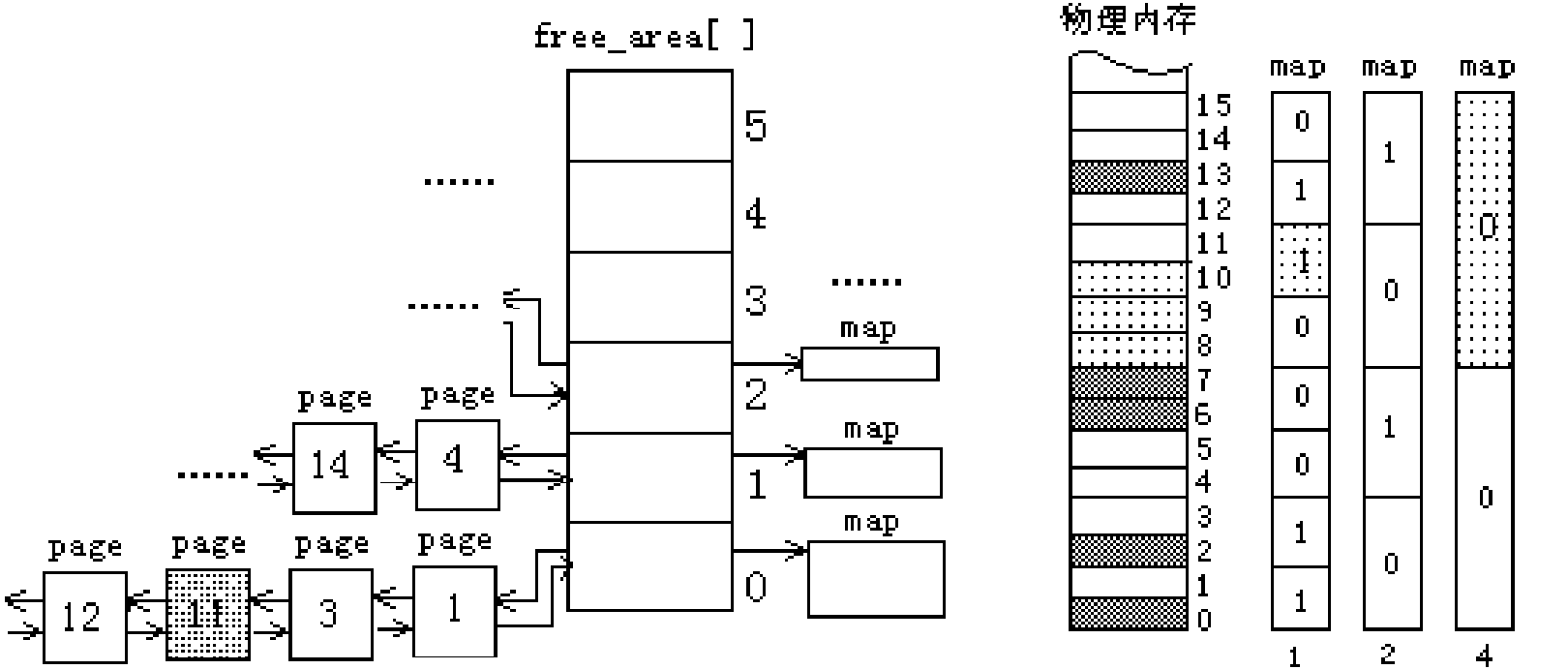
- 当两个伙伴都空闲时，则合并成一个更大的块
- 该过程一直进行，直到找不到可以合并的伙伴为止
- 位图用来描述伙伴的状态
  - 一对伙伴只使用一个位表示
    - ✓0：伙伴的状态一致，此时要么全空闲，要么全不（或部分）空闲
      - 如果全空闲，必然被合并了
      - 两种情况下，对应的块数据结构都不在此**free_area_t**结构中
    - ✓1：伙伴的状态不一致，此时必然有一个空闲、一个不空闲
      - 表示对应的块数据结构在此**free_area_t**结构的链表中

伙伴的位图管理示例



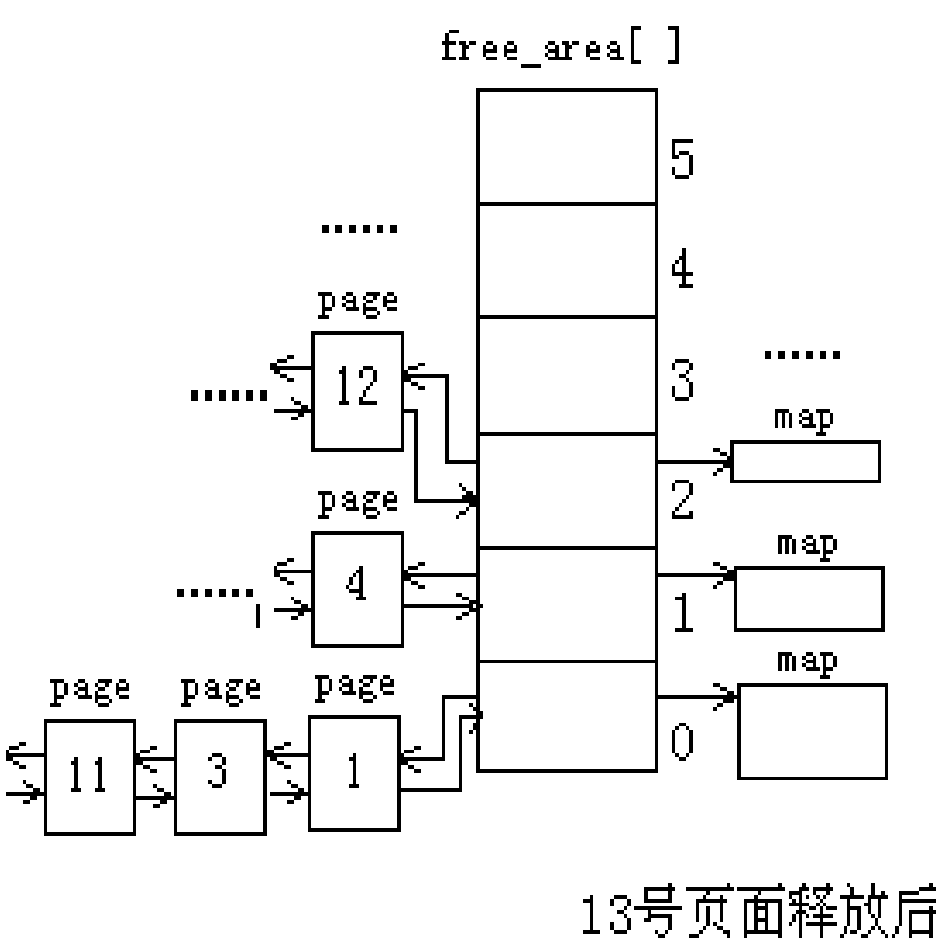
内存分配的Buddy算法

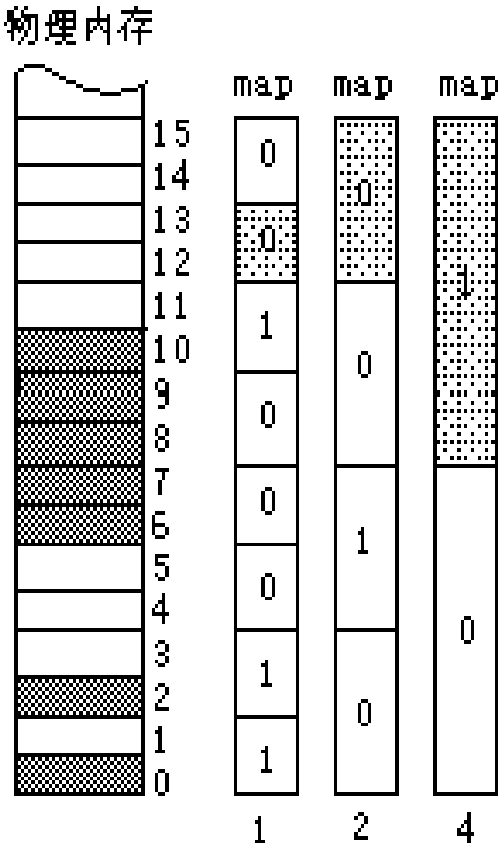
伙伴的位图管理示例



8、9、10页面分配后的示意

伙伴的位图管理示例


free_area[ ]
5
4
3
2
1
0
page
12
page
4
page
page
page
11
3
1
map
map
map
13号页面释放后


物理内存
15
14
13
12
11
10
9
8
7
6
5
4
3
2
1
0
map
map
map
0
0
1
0
0
0
1
0
1
0
1
0
1
0
1
0
1
2
4

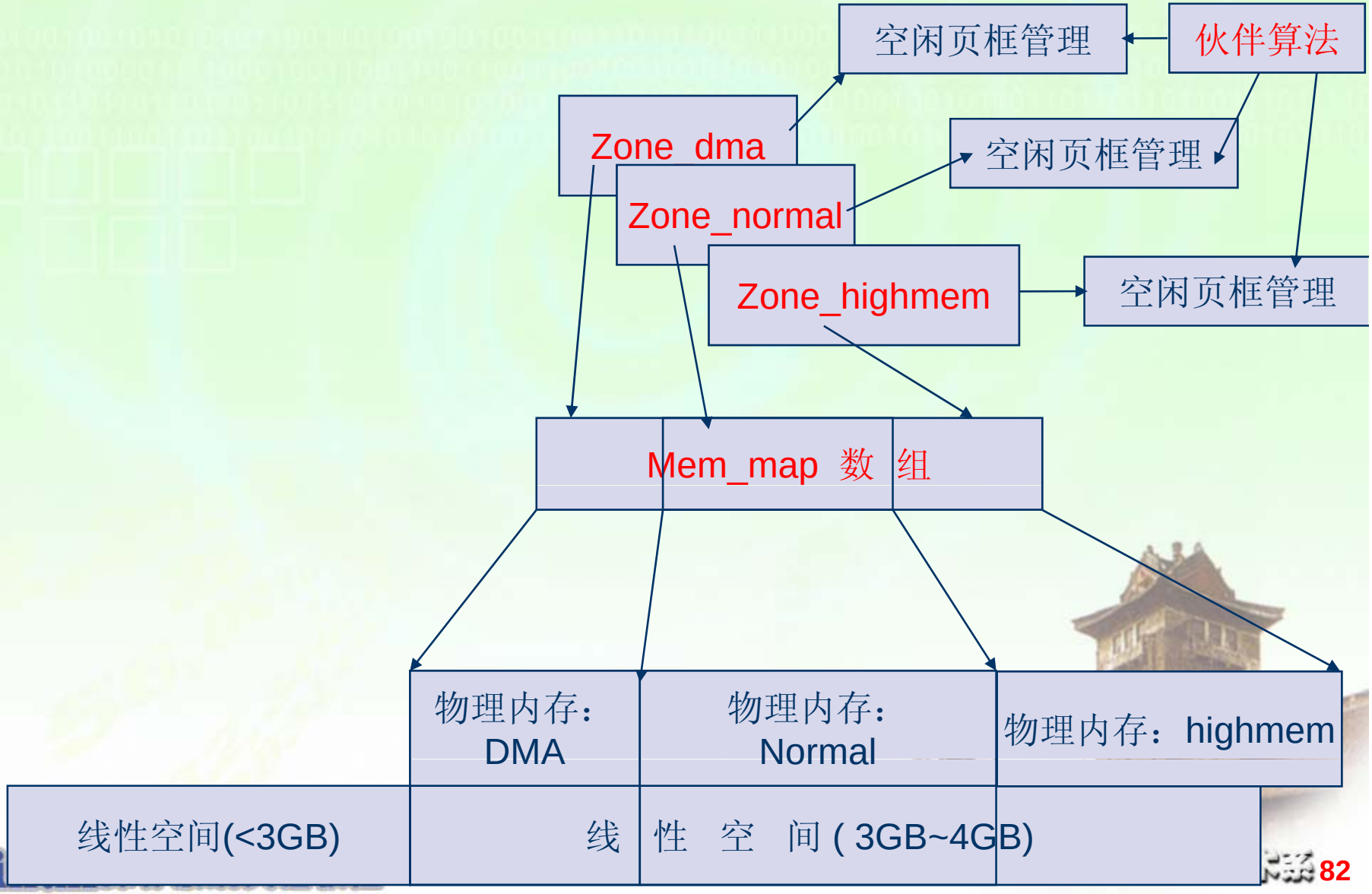
伙伴系统举例

- 伙伴系统的构建（假设有128MB的RAM）
  - 128MB最多可以分成2^{15}=32768个页框，2^{14}=16384个8KB（2页）的块或2^{13}=8192个16KB（4页）的块，直至64个大小为512个页的块
  - free_area[0]对应的位图由16384位组成，每对伙伴（大小为1个页框）对应其中的一位
  - free_area[1]对应的位图由8192位组成，每对伙伴块（大小为2页）对应其中的一位
  - …
  - free_area[9]对应的位图由32位组成，每对伙伴（大小为512页）对应其中的一位

伙伴系统举例

- 伙伴系统的使用（假设要请求一个大小为128个页框(0.5MB)的块
  - 算法先free_area[7]中检查是否有空闲块（块大小为128个页框）
  - 若没有，就到free_area[8]中找一个空闲块（块大小为256个页框）
    - ✓ 若存在这样的块，内核就把256个页框分成两等份，一半用作满足请求，另一半插入free_area[7]中
  - 如果在free_area[8]中也没有空闲块，就继续找free_area[9]中是否有空闲块
    - ✓ 若有，先将512分成伙伴，一个插入free_area[8]中，另一个进一步划分成伙伴，取其一插入free_area[7]中，另一个分配出去
  - 如果free_area[9]也没有空闲块，内存不够，返回一个错误信号

页框管理小结


空闲页框管理
伙伴算法
Zone_dma
空闲页框管理
Zone_normal
Zone_highmem
空闲页框管理
Mem_map 数 组
物理内存：DMA
物理内存：Normal
物理内存：highmem
线性空间(<3GB)
线 性 空 间 (3GB~4GB)

主要内容

- 背景知识
  - x86的分段机制
  - 物理存储管理
    - ✓ 页框管理
    - ✓ 内存区管理
    - ✓ 非连续存储区管理
  - 进程虚拟存储管理
  - slab分配器
- 实验内容
  - 统计系统和单个进程的缺页次数

内存区管理

- 基于页面的分配器不能满足多种要求
  - 内核中大量使用各种数据结构，大小从几个字节到几十上百k不等，都取整到2的幂次个页面那是完全不现实的
  - 早期内核的解决方法
    - ✓ 提供大小为2,4,8,16,...,131056字节的内存区域
    - ✓ 需要新的内存区域时，内核从伙伴系统申请页面，把它们划分成一个个区域，取一个来满足需求
    - ✓ 如果某个页面中的内存区域都释放了，页面就交回到伙伴系统

早期内存区管理的缺陷

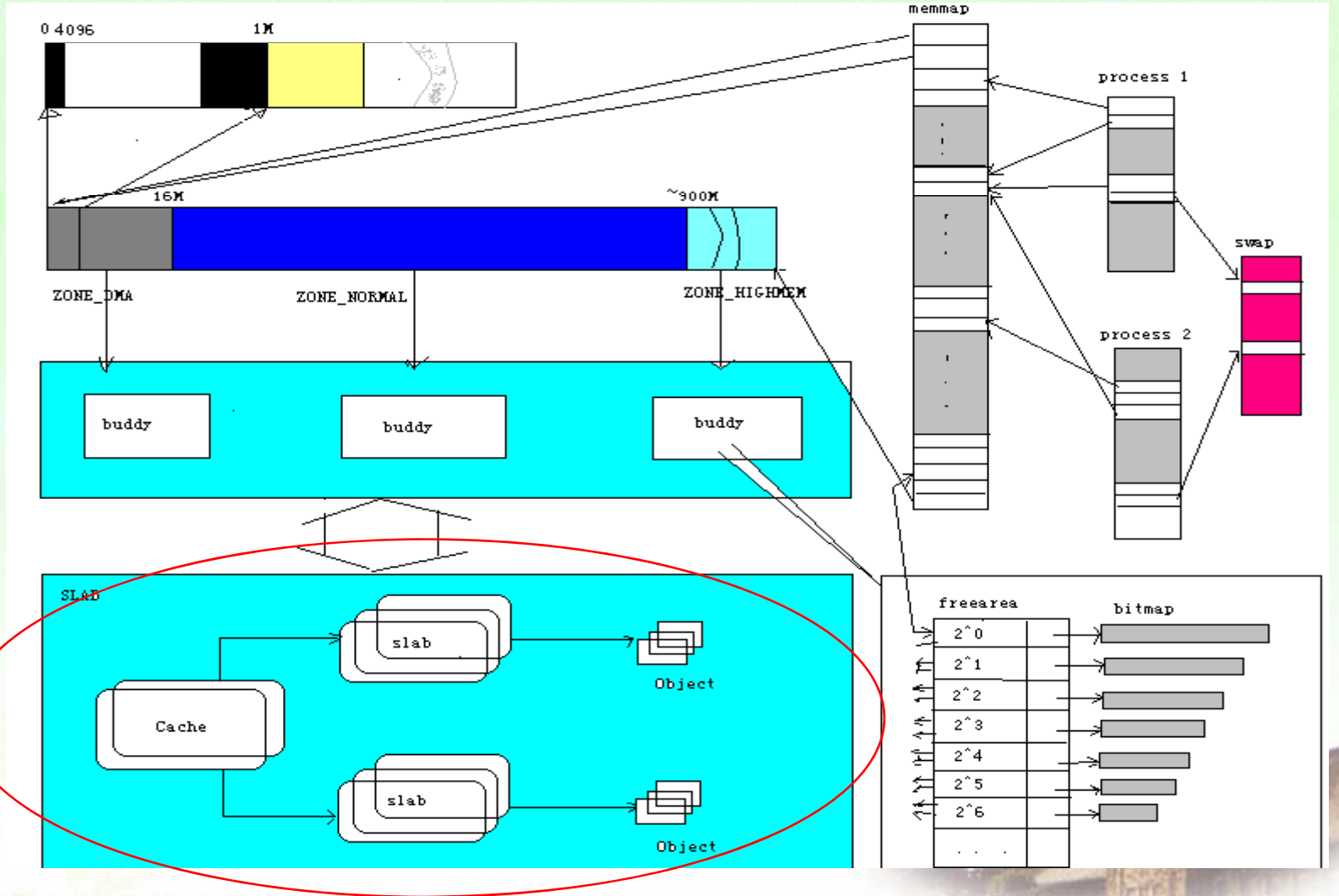
- 不同数据类型用不同的方法分配内存可能提高效率
  - 如需要初始化的数据结构，释放后可以暂存着，再分配时就不必初始化了
- 内核的函数常常重复地使用同一类型的内存区，缓存最近释放的对象可以加速分配和释放
- 对内存的请求可以按照请求频率来分类，频繁使用的类型使用专门的缓存，很少使用的可以使用通用缓存
- 使用2的幂次大小的内存区域时硬件高速缓存冲突的概率较大，有可能通过仔细安排内存区域的起始地址来减少硬件高速缓存冲突
- 缓存一定数量的对象可以减少对buddy系统的调用，从而节省时间并减少由此引起的硬件高速缓存污染

slab分配器

❖slab分配器的改进思想

- 将内存区看成对象
- 将对象按照类型分组成不同的高速缓存，每个高速缓存都是同种类型内存对象的一种“储备”
  - ✓ 例如当一个文件被打开时，存放相应所需的内存是从一个叫做filp的slab分配器的高速缓存中得到的，即每种对象类型对应一个高速缓存
- slab分配器通过伙伴系统分配页框
- 每个slab由一个或多个连续的页框组成，这些页框中包含已分配的对象，也包含空闲的对象

支持slab分配器的内存管理视图



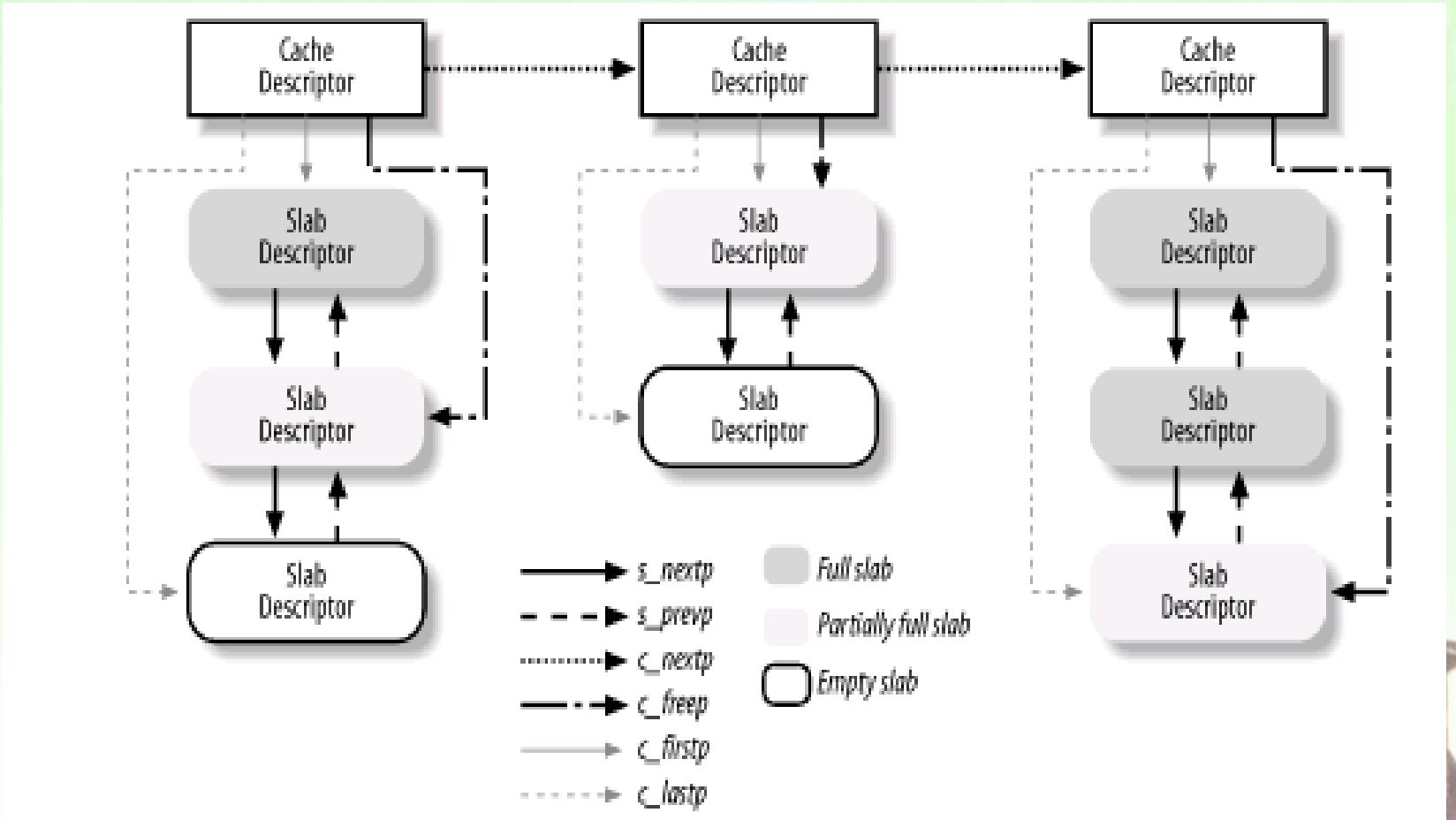
slab的状态与分配策略

- 每个slab有三种状态
  - 全满
    - ✓全满意味着slab中的对象全部已被分配出去
  - 半满
    - ✓半满介于两者之间
  - 全空
    - ✓全空意味着slab中的对象全部是可用的
- slab对象的分配策略
  - 优先从半满的slab满足这个请求
  - 否则从全空的slab中取一个对象满足请求
  - 如果没有空的slab则向buddy系统申请页面生成一个新的slab

slab分配器的基本数据结构

- 高速缓存描述符：**kmem_cache_t**
  - 链表结构
  - 包含三种slab双向循环链表
    - ✓ **slabs_full**
    - ✓ **slabs_partial**
    - ✓ **slabs_free**
- **slab描述符：slab_s**
- 对象描述符：**kmem_bufctl_t**

高速缓存描述符和slab描述符之间的关系



slab描述符

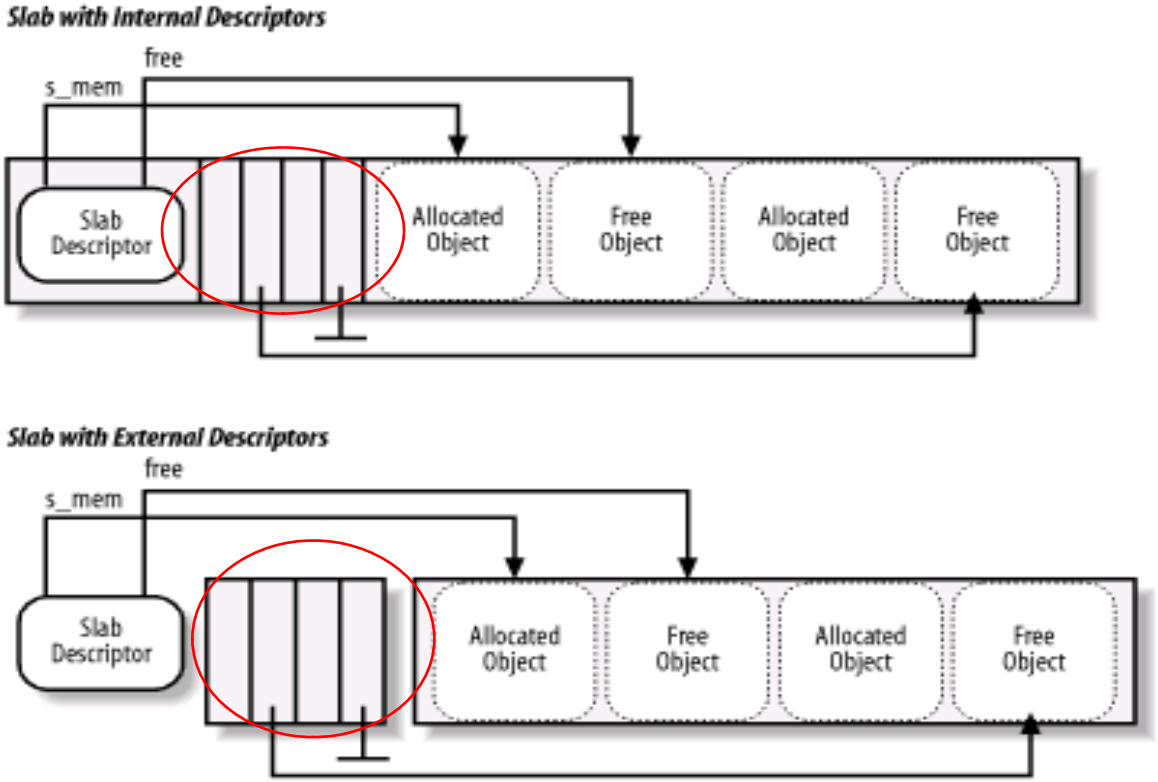
❖结构定义

```
struct slab{
   struct list_head   list;        /* 满、部分满或空链表 */
   unsigned long      colouroff;   /* slab着色的偏移量 */
   void               *s_mem;      /* 在slab中的第一个对象 */
   unsigned int       inuse;       /* 已分配的对象数*/
   kmem_bufctl_t      free;        /* 第一个空闲对象（如果有的话）*/
};
```

- 在高速缓存中根据类型不同分别存在不同的slab链表中
- 可存放在slab外，也可存放在slab内

对象描述符

- 结构定义：**kmem_bufctl_t**
  - 存放在数组中，位于相应的slab描述符之后
  - 只在对象空闲时有意义，包含的是下一个空闲对象在slab中的下标



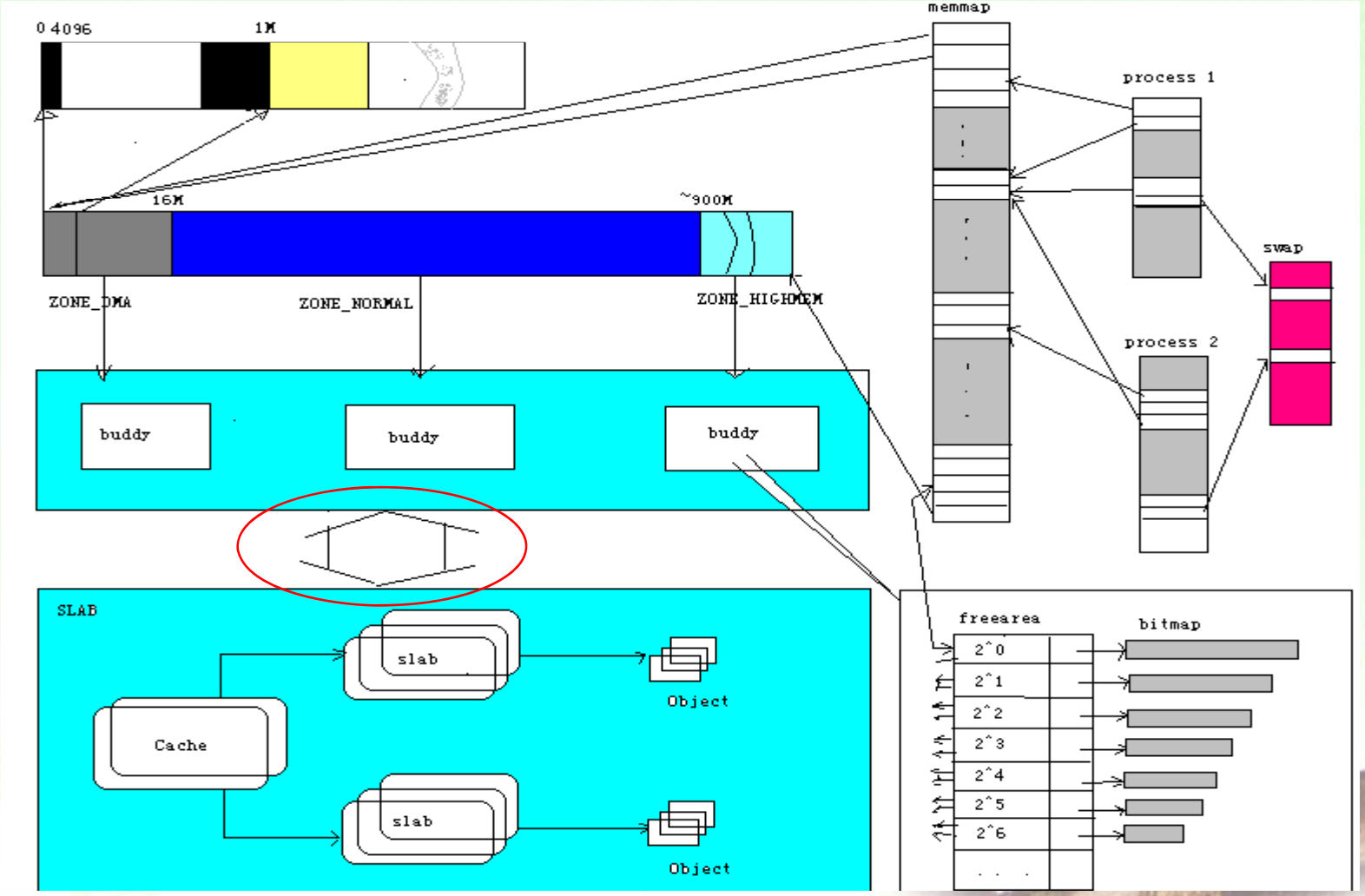
高速缓冲器

- 分类
  - 普通高速缓存(general cache)
    - slab分配器自己使用
    - 根据大小分配内存
      - 26个，2组（一组用于DMA分配，另一组用于常规分配）
      - 每组13个，大小从2^5=32个字节，到2^{17}=132017个字节
  - 专用高速缓存(special cache)
    - 内核其余部分使用
    - 根据类型分配

普通高速缓存结构

```
/* Size description struct for general caches. */
typedef struct cache_sizes {
        size_t          cs_size;
        kmem_cache_t*cs_cachep;
        kmem_cache_t*cs_dmacachep;
} cache_sizes_t;

static cache_sizes_t cache_sizes[] = {
#if PAGE_SIZE == 4096
        {   32,   NULL, NULL},
#endif
        {   64,   NULL, NULL},
        {  128,   NULL, NULL},
        {  256,   NULL, NULL},
        {  512,   NULL, NULL},
        { 1024,   NULL, NULL},
        { 2048,   NULL, NULL},
        { 4096,   NULL, NULL},
        { 8192,   NULL, NULL},
        {16384,   NULL, NULL},
        {32768,   NULL, NULL},
        {65536,   NULL, NULL},
        {131072,  NULL, NULL},
        {    0,   NULL, NULL}
};
```

slab分配器与伙伴系统的接口



slab分配器与伙伴系统的接口

❖获取一组连续的空闲页框：**kmem_getpages()**

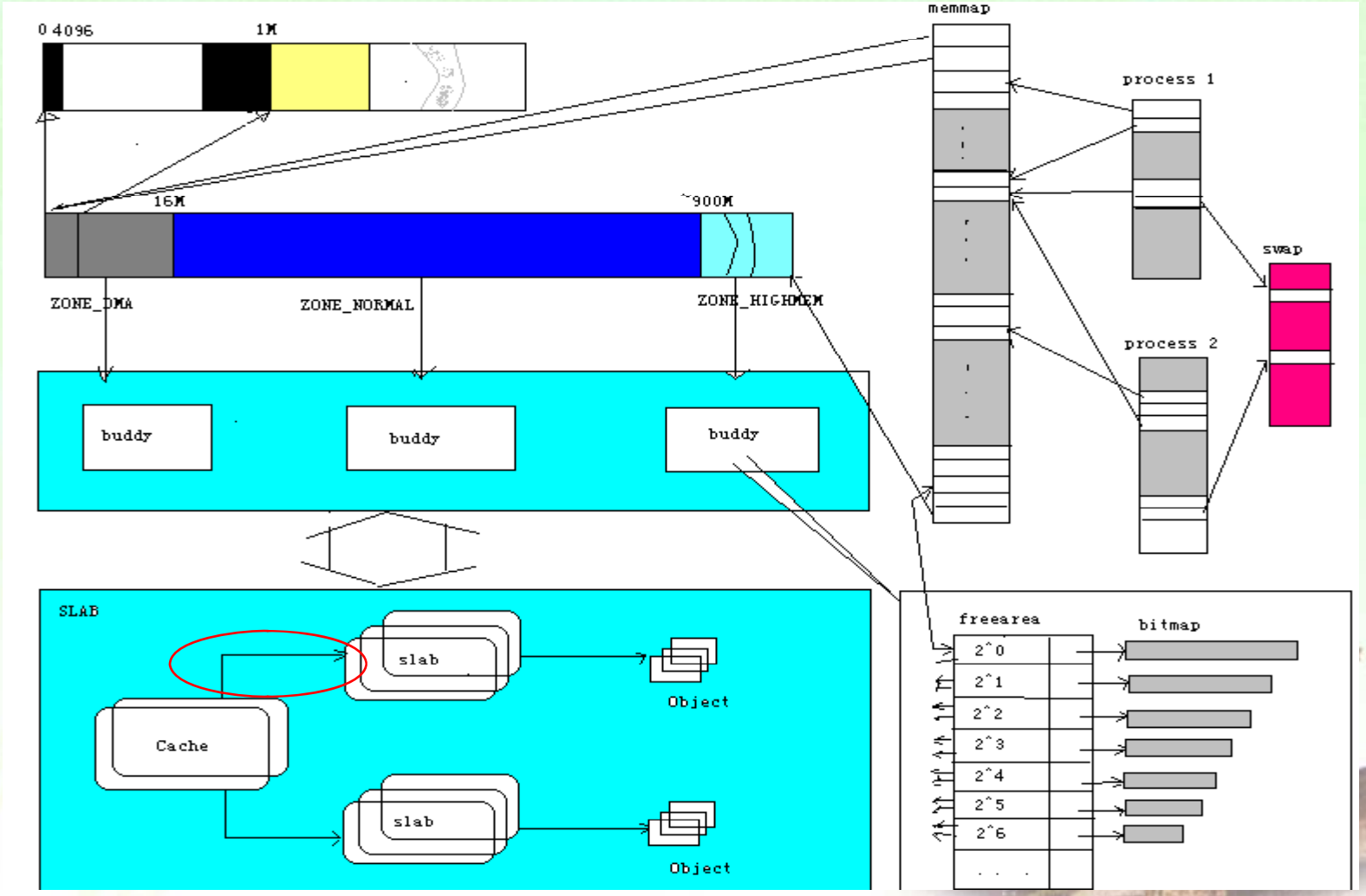
```
/* Interface to system's page allocator. No need to hold the cache-lock.
 */
static inline void * kmem_getpages (kmem_cache_t *cachep, unsigned long flags)
{
	void *addr;

	/*
	 * If we requested dmaable memory, we will get it. Even if we
	 * did not request dmaable memory, we might get it, but that
	 * would be relatively rare and ignorable.
	 */
	flags |= cachep->gfpflags;
	addr = (void*) __get_free_pages(flags, cachep->gfporder);
	/* Assume that now we have the pages no one else can legally
	 * messes with the 'struct page's.
	 * However vm_scan() might try to test the structure to see if
	 * it is a named-page or buffer-page.  The members it tests are
	 * of no interest here.....
	 */
	return addr;
}
```

slab分配器与伙伴系统的接口

❖释放分配给slab分配器的页框：**kmem_freepages()**

```
/* Interface to system's page release. */
static inline void kmem_freepages (kmem_cache_t *cachep, void *addr)
{
        unsigned long i = (1<<cachep->gfporder);
        struct page *page = virt_to_page(addr);

        /* free_pages() does not clear the type bit - we do that.
         * The pages have been unlinked from their cache->slab,
         * but their 'struct page's might be accessed in
         * vm_scan(). Shouldn't be a worry.
         */
        while (i--) {
                PageClearSlab(page);
                page++;
        }
        free_pages((unsigned long)addr, cachep->gfporder);
}
```

slab的分配与释放


0 4096
1M
16M
~900M
ZONE_DMA
ZONE_NORMAL
ZONE_HIGHMEM
buddy
buddy
buddy
SLAB
slab
Object
Cache
slab
Object
memmap
process 1
process 2
swap
freearea
bitmap
2^0
2^1
2^2
2^3
2^4
2^5
2^6

slab的分配—kmem_cache_grow()

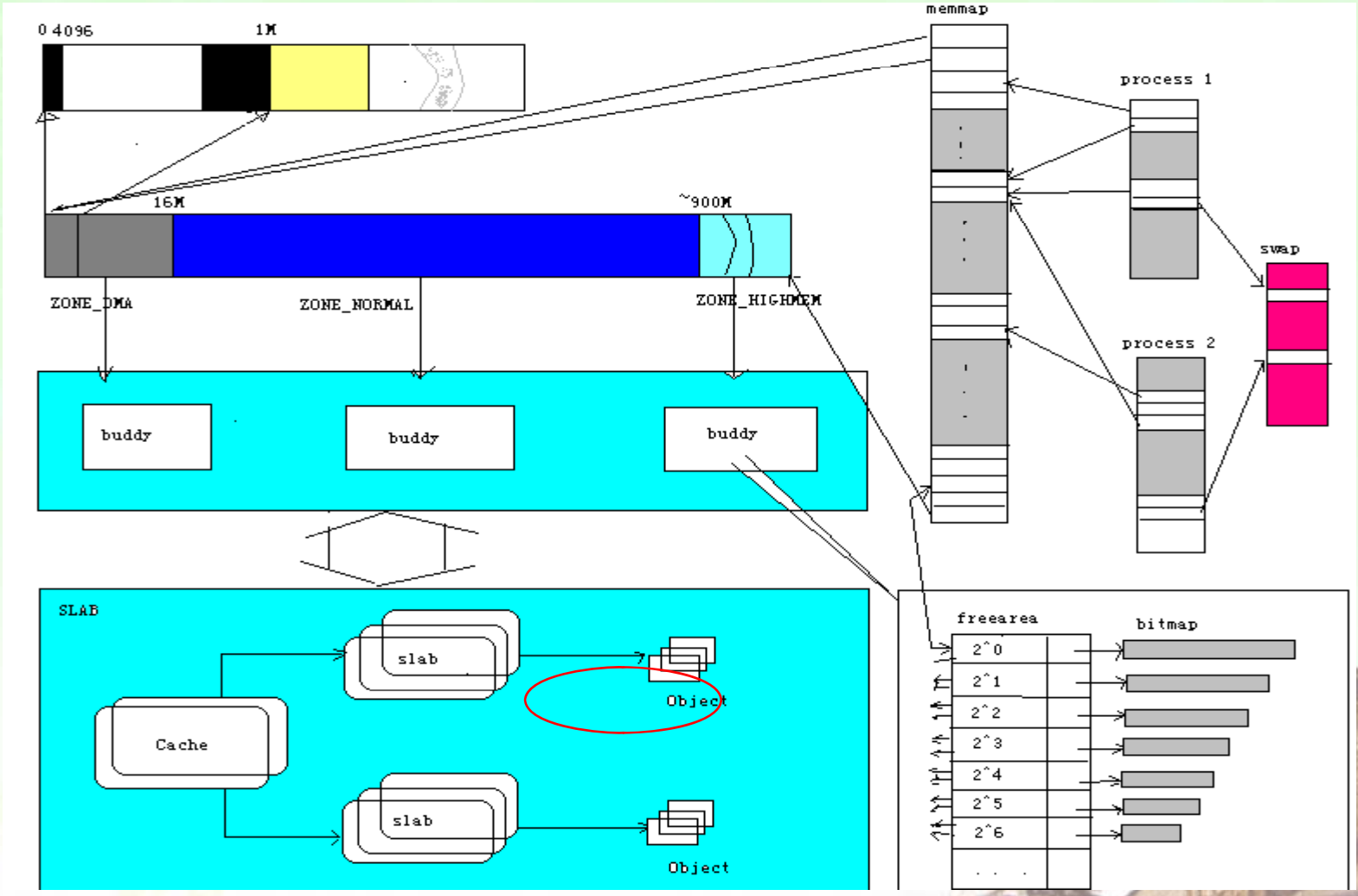
- 分配时机
  - 已发出分配新对象的请求，并且高速缓存不包含任何空闲对象
- 处理过程
  - **kmem_getpages()**为slab分得页框
    - ✓ **kmem_cache_slabmgmt()获得新slab描述符**
  - **kmem_cache_init_objs()**为新的slab中的对象申请构造方法
  - 扫描所有页框描述符
    - ✓ **高速缓存描述符地址存页描述符中的list->next**
    - ✓ **slab描述符地址存页描述符中的list->prev**
    - ✓ **设置页描述符的PG_slab标志**
  - 将slab描述符加到全空slab链表中

slab的释放—kmem_slab_destroy

- 释放时机
  - slab高速缓存有太多的空闲对象
  - 被周期性调用的定时器函数确定是否有完全未使用的slab

```
void kmem_slab_destroy(kmem_cache_t *cachep, slab_t *slabp)
{
   if (cachep->dtor) { //与高速缓存相关的析构方法的指针
      int i;
      for (i = 0; i < cachep->num; i++) {
         void* objp = & slabp->s_mem[cachep->objsize*i];
         (cachep->dtor)(objp, cachep, 0);
      }
   }
   kmem_freepages(cachep, slabp->s_mem - slabp->colouroff);
   if (OFF_SLAB(cachep))
      kmem_cache_free(cachep->slabp_cache, slabp);
}
```

高速缓存中的对象管理


0 4096
1M
16M
~900M
ZONE_DMA
ZONE_NORMAL
ZONE_HIGHMEM
buddy
buddy
buddy
SLAB
slab
Object
Cache
slab
Object
memmap
process 1
process 2
swap
freearea
bitmap
2^0
2^1
2^2
2^3
2^4
2^5
2^6

高速缓存中的对象管理

- 从高速缓存中分配一个内存对象
  - kmem_cache_alloc()

kmem_cache_alloc()中的代码片段：

```
void * objp;
slab_t * slabp;
struct list_head * entry;
local_irq_save(save_flags);
entry = cachep->slabs_partial.next;
if (entry != NULL & cachep->slabs_partial) {
    entry = cachep->slabs_free.next;
    if (entry == & cachep->slabs_free)
        goto alloc_new_slab;
    list_del(entry);
    list_add(entry, & cachep->slab_partials);
}
slabp = list_entry(entry, slab_t, list);
```

查找空闲对象的slab，若不存在，则分配一个新的slab

高速缓存中的对象管理

❖从高速缓存中分配一个内存对象

```
slabp->inuse++;
            ...
objp = & slabp->s_mem[slabp->free * cachep->objsize];
            …
slabp->free = ((kmem_bufctl_*)(slabp+1))[slabp->free];
if (slabp->free == BUFCTL_END) {
   list_del(&slabp->list);
   list_add(&slabp->list, &cachep->slabs_full);
}
            …
local_irq_restore(save_flags);
return objp;
```

高速缓存中的对象管理

❖从高速缓存中释放一个内存对象

➢ kmem_cache_free()

kmem_cache_free()中的代码片段：

```
slab_t * slabp;
unsigned int objnr;
local_irq_save(save_flags);
slabp = (slab_t *) mem_map[_ _pa(objp) >> PAGE_SHIFT].list.prev;
        …
objnr = (objp - slabp->s_mem) / cachep->objsize;
((kmem_bufctl_t *)(slabp+1))[objnr] = slabp->free;
slabp->free = objnr;
```

计算出slab描述符地址，在为slab分配页框时，描述符地址存于页框描述符的list.prev中

导出对象描述符，将对象追加到空闲对象链表的首部

数据结构：

slabp->free:指向slab内第一个空闲对象

对象描述符数组紧挨着slab描述符，表项指向下一个空闲对象

高速缓存中的对象管理

- 从高速缓存中释放一个内存对象

kmem_cache_free()中的代码片段（接上）：

```
if (--slabp->inuse == 0) { /* slab is now fully free */
    list_del(&slabp->list);
    list_add(&slabp->list, &cachep->slabs_free);
} else if (slabp->inuse+1 == cachep->num) { /* slab was full */
    list_del(&slabp->list);
    list_add(&slabp->list, &cachep->slabs_partial);
}
local_irq_restore(save_flags);
return;
```

最后检查slab是否需要移动另一个链表中

高速缓存中的对象管理

- 高速缓存中的通用对象管理
  - 从普通高速缓存中分配
    - ✓从通用缓存中申请大小为size的一段连续物理内存
    - ✓返回指向这段内存线性地址的指针

```
void * kmalloc (size_t size, int flags)
```

  - 从普通高速缓存中释放
    - ✓释放由一个kmalloc()函数申请的内存

```
void kfree (const void *objp)
```

高速缓存中的对象管理

❖普通高速缓存分配函数kmalloc()

```
void * kmalloc(size_t size, int flags) {
    cache_sizes_t *csizep = cache_sizes;
    kmem_cache_t * cachep;
    for (; csizep->cs_size; csizep++) {
        if (size > csizep->cs_size)
            continue;
        if (flags & _ _GFP_DMA)
            cachep = csizep->cs_dmacahep;
        else
            cachep = csizep->cs_cachep;
        return _  _kmem_cache_alloc(cachep, flags);
    }
    return NULL;
}
```

高速缓存中的对象管理

❖普通高速缓存释放函数kfree()

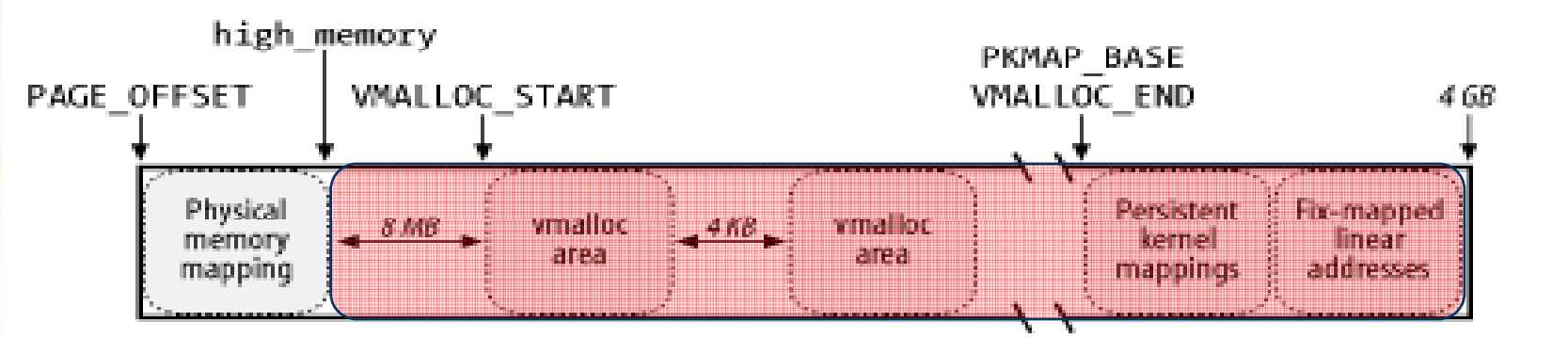
```
void kfree(const void *objp)
{
    kmem_cache_t * c;
    unsigned long flags;
    if (!objp)
        return;
    local_irq_save(flags);
    c = (kmem_cache_t *) mem_map[_ _pa(objp) >> PAGE_SHIFT].list.next;
    _ _kmem_cache_free(c, (void *) objp);
    local_irq_restore(flags);
}
```

主要内容

- 背景知识
  - x86的分段机制
  - 物理存储管理
    - ✓ 页框管理
    - ✓ 内存区管理
    - ✓ 非连续存储区管理
  - 进程虚拟存储管理
  - slab分配器
- 实验内容
  - 统计系统和单个进程的缺页次数

高端内存页框的内核映射

- 起始地址存放在**high_memory**中，不直接映射在内核线性地址空间的第4个GB，内核不能直接访问
  - 返回所分配页框线性地址的页分配器不适用于高端内存
    - ✓ __get_free_pages()在高端内存分配页框成功后，不能返回其线性地址
  - 高端内存页框分配只能通过alloc_pages()或alloc_page()
    - ✓ 返回第一个被分配的页框的页描述符地址
  - 内核线性地址空间的最后128MB的一部分用于映射高端内存页框

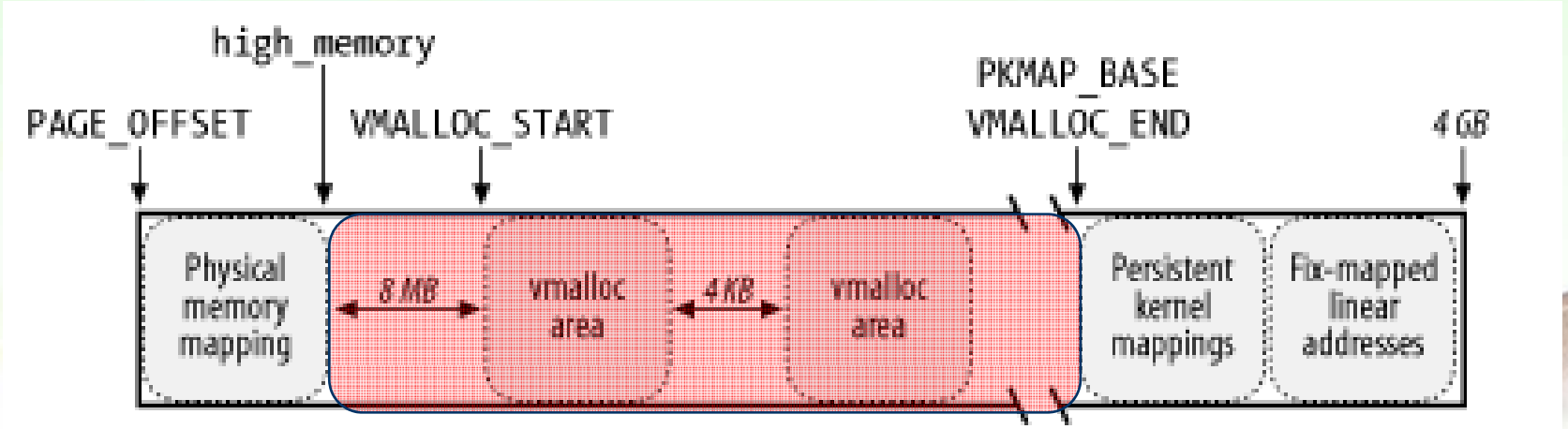


高端内存页框的内核映射方法

- 永久内核映射
  - 在空闲页表项不存在（即高端内存上没有页表项可用做页框的窗口）时会阻塞当前进程
  - 不能用于中断处理程序和可延迟函数
- 临时内核映射
  - 使用临时内核映射的内核控制路径必须保证当前没有其他的内核控制路径在使用同样的映射，这意味着内核控制路径永远不能被阻塞
  - 只有很少的临时内核映射可以同时建立
- 非连续内存分配

非连续存储区管理概述

- 必要性
  - 把线性空间映射到一组连续的页框是很好的选择
  - 有时候不得不将线性空间映射到一组不连续的页框
    - ✓优点：避免碎片
- 为非连续内存区保留的线性地址空间
  - **VMALLOC_START~VMALLOC_END**



非连续存储区的基本操作

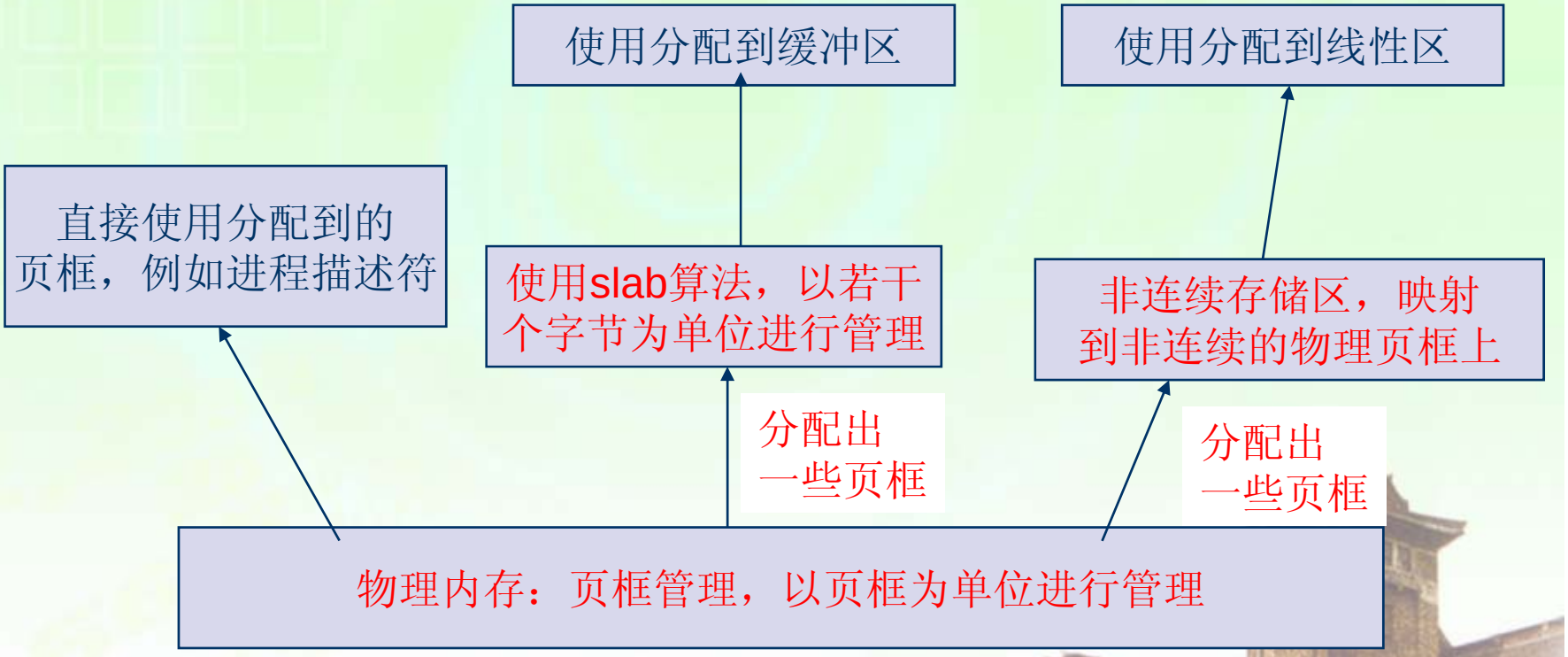
❖非连续存储区描述符

```
struct vm_struct {
        unsigned long flags;
        void * addr;
        unsigned long size;
        struct vm_struct * next;
};
```

❖相关函数

- get_vm_area(): 创建vm_struct描述符
- vmalloc(): 分配非连续存储器区
  - ✓调用get_vm_area创建一个新的vm_struct结构
  - ✓调用vmalloc_area_pages请求非连续的页框
- vfree(): 释放非连续内存区
  - ✓调用vmfree_area_pages释放页框

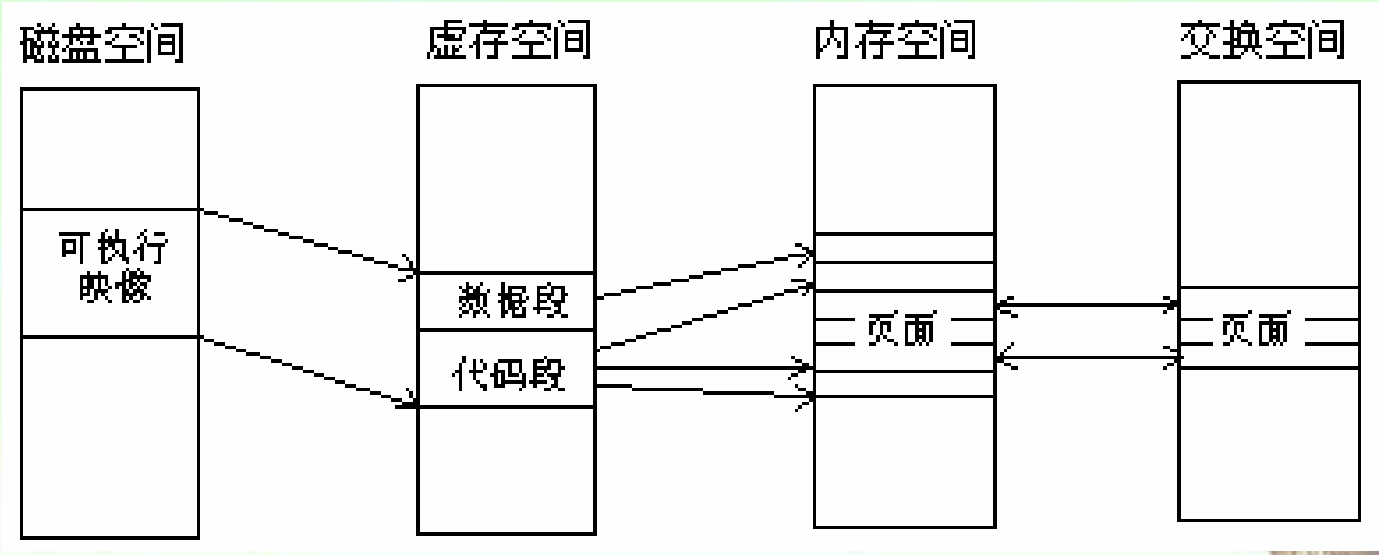
物理存储管理小结


使用分配到缓冲区
使用分配到线性区
直接使用分配到的
页框，例如进程描述符
使用slab算法，以若干
个字节为单位进行管理
非连续存储区，映射
到非连续的物理页框上
分配出
一些页框
分配出
一些页框
物理内存：页框管理，以页框为单位进行管理

主要内容

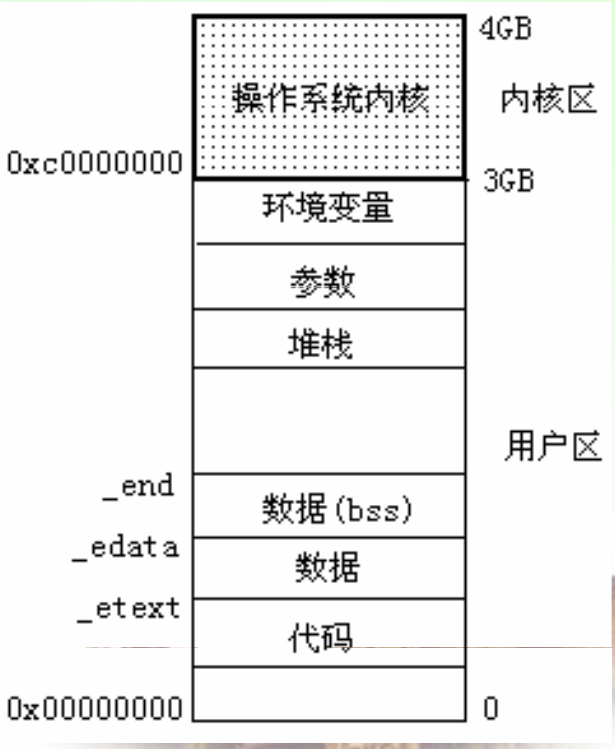
- 背景知识
  - x86的分段机制
  - 物理存储管理
  - 进程虚拟存储管理
    - ✓ 进程的地址空间
    - ✓ 内存描述符
    - ✓ 线性区描述符
    - ✓ 线性区的处理
    - ✓ 虚存与实存的映射
- 实验内容
  - 统计系统和单个进程的缺页次数

Linux进程的地址映射



进程虚拟空间

- Linux把进程虚拟空间分成内核区和用户区两部分
  - 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
    - ✓ 一个进程所需的虚拟空间中的各个部分未必连续，这通常会形成若干离散的虚存“区间”(VM area)
    - ✓ 一个虚拟“区间”是进程虚拟空间的一部分，这部分的虚拟空间是连续的并且有相同的一些属性



内核态和用户态分配内存的差别

- 内核中的函数以直接了当的方式获得动态内存
  - 内核是操作系统中优先级最高的成分，内核信任自己
  - 采用页面级内存分配和小内存分配
- 用户态进程分配内存时
  - 请求被认为是不紧迫的，用户进程不可信任
  - 没有立即获得实际的物理页框，而仅仅获得对一个新的线性地址区间的使用权
  - 这个线性地址区间会成为进程地址空间的一部分，称作线性区或内存区域(memory region)

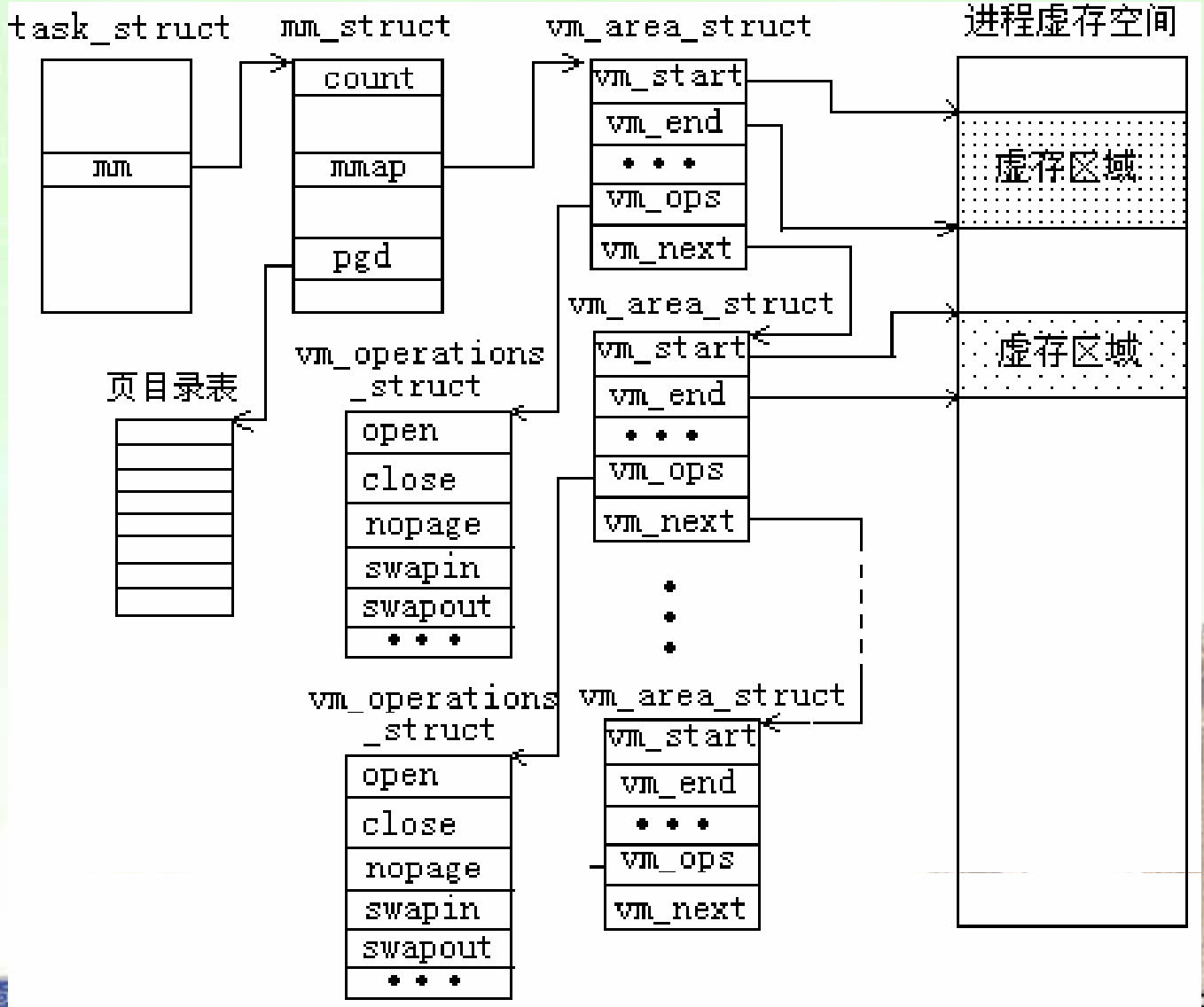
进程地址空间

- 进程在访问某个线性空间之前，必须先获得许可
  - 一个进程的地址空间是由允许该进程访问的全部线性地址组成
  - 内核使用线性区资源来表示线性地址空间
  - 每个线性区由起始线性地址、长度和一些存取权限描述

进程内存组织相关数据结构

- 每个进程有且只有一个**mm_struct**结构，描述进程的虚拟内存
- **vm_area_struct**结构描述进程的虚拟内存地址区域（线性区）
  - 每个进程可以有多个**VM area**
  - 对页错误处理有同一规则的进程虚拟内存空间部分，如共享库、堆栈
- **page**结构描述一个物理页，系统保证跟踪到每一个物理页

进程的内存组织


task_struct
mm_struct
vm_area_struct
进程虚存空间
mm
count
mmap
pgd
vm_start
vm_end
• • •
vm_ops
vm_next
虚存区域
虚存区域
vm_area_struct
vm_start
vm_end
• • •
vm_ops
vm_next
页目录表
vm_operations
_struct
open
close
nopage
swapin
swapout
• • •
vm_operations
_struct
open
close
nopage
swapin
swapout
• • •
vm_area_struct
vm_start
vm_end
• • •
vm_ops
vm_next

主要内容

- 背景知识
  - x86的分段机制
  - 物理存储管理
  - 进程虚拟存储管理
    - ✓ 进程的地址空间
    - ✓ 内存描述符
    - ✓ 线性区描述符
    - ✓ 线性区的处理
    - ✓ 虚存与实存的映射
- 实验内容
  - 统计系统和单个进程的缺页次数

内存描述符

- **struct mm_struct[include/linux/sched.h]**
  - 所有内存描述符放在由**mmlist**字段组成的双向链表中
  - 链表第一个元素是**init_mm**，是进程**0**的内存描述符
  - 关键字段说明
    - ✓ **mm_users：共享mm_struct的轻量级进程的个数**
    - ✓ **mm_count：内存描述符的住使用计数器**
      - **mm_users**次使用计数器的所有用户在**mm_count**中只作为一个单位
      - 每当**mm_users**减少时，内核都要检查它是否变为**0**，若是，则接触这个内存描述符
      - 如果把内核描述符暂时借给一个内核线程，则增加**mm_count**

```
struct mm_struct {
    struct vm_area_struct * mmap;       /* list of VMAs */
    rb_root_t mm_rb;
    struct vm_area_struct * mm
    pgd_t * pgd;
    atomic_t mm_users;
    atomic_t mm_count;
    int map_count;
    struct rw_semaphore mmap_sem;
    spinlock_t page_table_lock;         /* Protects task page tables and mm->

    struct list_head mmlist;
                        * toge
                        * by mm
                        */

    unsigned long start_code, end_code, start_data, end_data;
    unsigned long start_brk, brk, start_stack;
    unsigned long arg_start, arg_end, env_start, env_end;
    unsigned long rss, total_vm, locked_vm;
    unsigned long def_flags;
    unsigned long cpu_vm_mask;
    unsigned long swap_address;

    unsigned dumpable:1;

    /* Architecture-specific MM context */
    mm_context_t context;
} ? end mm_struct ? ;
```

页全局目录

nap和mm_rb是两个不同的数据结构，但是包含了相同的东西:进程地址空间中所有的memory areas

前者使用链表存储areas

后者用红黑树存储areas

所有的mm_struct结构通过mmlist域链接在一个双向链表上。这个链表的第一个元素init_mm是idle进程的mm_struct结构

内核线程的内存描述符

- 内核线程仅运行在内核态，不会访问低于TASK_SIZE(等于0xc0000000)的地址
- 进程描述符中的两个内存描述符mm和active_mm
  - mm：指向进程拥有的内存描述符
  - active_mm：指向进程运行时所使用的内存描述符
  - 对普通进程而言，这两个字段存放相同的指针
  - 对内核线程而言
    - ✓ 由于不拥有任何内核描述符，因此，mm字段总为NULL
    - ✓ 当内核线程运行时，active_mm被初始化为前一个运行进程的active_mm的值

内存描述符的分配

- 在fork()通过调用copy_mm()从父进程拷贝内存描述符
- mm_struct数据结构本身的空间通过宏allocate_mm() 从mm_cachep指向的slab缓存中分配得到的
- 如果父进程在fork()创建子进程时，通过一些标志指明要和子进程共享地址空间。那么，只需要
  - childtask->mm = parent->mm

```
if(clone_flags & CLONE_VM){
        /*
         * current 是父进程而tsk在fork()执行期间是子进程
         */
        atomic_inc(&current->mm->mm_users);
        tsk->mm= current ->mm;
}
```

内存描述符的释放

- 在进程退出时，exit_mm()函数被调用
  - 首先做一些常规清除工作，更新一些内核全局统计数据
  - 接着调用mmput()，减少内存描述符的mm_users域
    - 如果mm_users域变成0，就调用mmdrop()函数减mm_count域
      - 如果mm_count域变成0，就由free_mm()宏调用kmem_cache_free()函数把mm_struct返还给mm_cachp指向slab缓存

主要内容

- 背景知识
  - x86的分段机制
  - 物理存储管理
  - 进程虚拟存储管理
    - ✓ 进程的地址空间
    - ✓ 内存描述符
    - ✓ 线性区描述符
    - ✓ 线性区的处理
    - ✓ 虚存与实存的映射
- 实验内容
  - 统计系统和单个进程的缺页次数

线性区（memory region）

- 由vm_area_struct结构体描述
  - 描述地址空间内连续区间上的一个独立内存范围
  - 线性区的开始和结束都必须4KB对齐
  - 进程只能访问某个有效的线性区
    - ✓ 如果进程试图访问一个有效的area之外的地址或者用不正确的方式访问一个有效的area，内核将通过段异常(segmentation fault)杀死这个进程
  - 进程获得新线性区的一些典型情况
    - ✓ 刚刚创建的新进程
    - ✓ 使用exec系统调用装载一个新的程序运行
    - ✓ 将一个文件（或部分）映射到进程地址空间中
    - ✓ 当用户堆栈不够用的时候，扩展堆栈对应的线性区
    - ✓ ……

线性区可包含内容

- 可执行文件代码段（*.text section*）的内存映射
- 数据段（ *.data section*）的内存映射
- 零页面（ *.bss section* ）的内存映射用来包含未初始化的全局变量
- 为库函数和链接器附加的代码、数据、bss段
- 文件的内存映射
- 共享内存的映射
- 匿名内存区域的映射，如通过**malloc()**函数申请的内存区域

线性区描述符vm_area_struct

- 描述一段给定的内存区间
- 区间中的地址都有同样的属性，如同样的存取权限和相关的操作函数
- 用这个结构可以表示各种线性区，如映射可执行的二进制代码的线形区、用作用户态堆栈的线性区等等

```
/*
 * This struct defines a memory VMM memory area. There is one of these
 * per VM-area/task.  A VM area is any part of the process virtual memory
 * space that has a special rule for the page-fault handlers (ie a shared
 * library, the executable area etc).
 */
struct vm_area_struct {
    struct mm_struct * vm_mm;    /* The address space we belong to. */
    unsigned long vm_start;      /* Our start address within vm_mm. */
    unsigned long vm_end;        /* The first byte after our end address
                                    within vm_mm. */

    /* linked list of VM areas per task, sorted by address */
    struct vm_area_struct *vm_next;

    pgprot_t vm_page_prot;       /* Access permissions of this VMA. */
    unsigned long vm_flags;      /* Flags, listed below. */

    rb_node_t vm_rb;

    /*
     * For areas with an address space and backing store,
     * one of the address_space->i_mmap{,shared} lists,
     * for shm areas, the list of attaches, otherwise unused.
     */
    struct vm_area_struct *vm_next_share;
    struct vm_area_struct **vm_pprev_share;

    /* Function pointers to deal with this struct. */
    struct vm_operations_struct * vm_ops;

    /* Information about our backing store: */
    unsigned long vm_pgoff;      /* Offset (within vm_file) in PAGE_SIZE
                                    units, *not* PAGE_CACHE_SIZE */
    struct file * vm_file;       /* File we map to (can be NULL). */
    unsigned long vm_raend;      /* XXX: put full readahead info here. */
    void * vm_private_data;      /* was vm_pte (shared mem) */
} ?  end vm_area_struct ?  ;
```

vm_area_struct结构体字段说明

- **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**
  - 指向下一个vm_area_struct结构体，所有vm_area_struct结构体链接成一个单向链表
  - 链表首地址由mm_struct中成员项mmap指出
- **vm_ops**
  - 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

vm_area_struct结构体字段说明

- **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

线性区访问权限

- **vm_flags**
  - 描述有关这个线性区全部页的信息，如
    - ✓ 进程访问每个页的权限
    - ✓ 线性区的增长方式

标　志	对VMA及其页面的影响
VM_READ	页面可读取
VM_WRITE	页面可写
VM_EXEC	页面可执行
VM_SHARED	页面可共享
VM_MAYREAD	VM_READ标志可被设置
VM_MAYWRITE	VM_WRITE标志可被设置
VM_MAYEXEC	VM_EXEC标志可被设置
VM_MAYSHARE	VM_SHARE标志可被设置
VM_GROWSDOWN	区域可向下增长
VM_GROWSUP	区域可向上增长
VM_SHM	区域可用作共享内存
VM_DENYWRITE	区域映射一个不可写文件
VM_EXECUTABLE	区域映射一个可执行文件
VM_LOCKED	区域中的页面被锁定
VM_IO	区域映射设备I/O空间
VM_SEQ_READ	页面可能会被连续访问
VM_RAND_READ	页面可能会被随机访问
VM_DONTCOPY	区域不能在fork()时被拷贝
VM_DONTEXPAND	区域不能通过mremap()增加
VM_RESERVED	区域不能被换出
VM_ACCOUNT	该区域是一个记账VM对象
VM_HUGETLB	区域使用了hugetlb页面
VM_NONLINEAR	该区域是非线性映射的

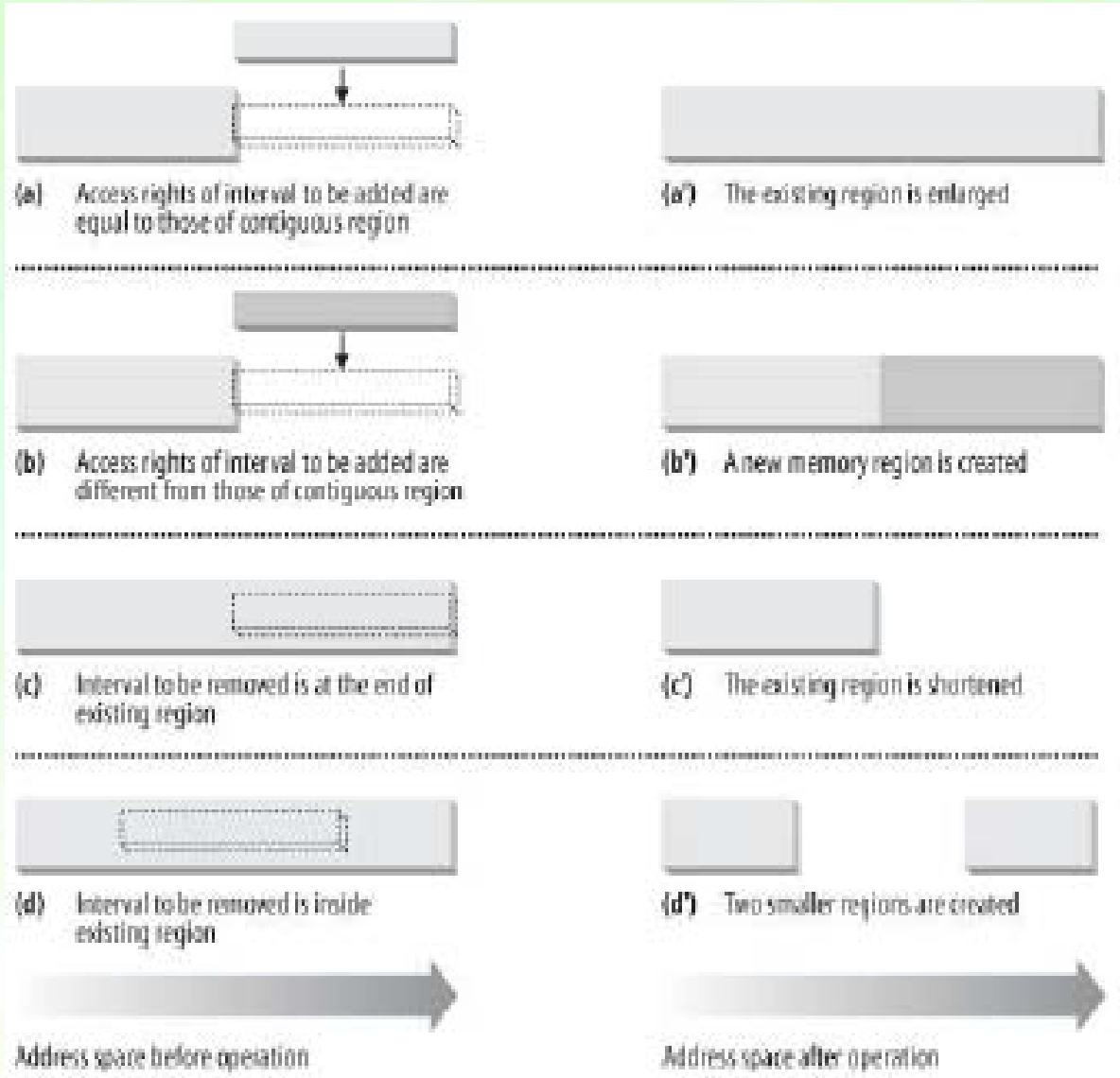
线性区基本操作函数

- **void open(struct vm_area_struct *area)**
  - 将指定的内存区域被加入到一个线性地址空间
- **void close(strcut vm_area_struct *area)**
  - 将指定的内存区域从地址空间删除
- **struct page *nopage(struct vm_area_struct *area, unsigned long address, int unsed)**
  - 访问的页不在物理内存时，该函数被页错误处理程序调用
- **int populate(strcut vm_area_struct *area, unsigned long addr, unsigned long len, pgprot_t, prot, unsigned long pgoff, int nonblock)**
  - 被系统调用remap_pages()调用来为将要发生的缺页重大预映射一个新映射

线性区维护

- 线性区从来不重叠
- 进程中每个单独区域对应一个不同内存区，如
  - 堆栈、二进制代码、全局变量、文件映射等等
- 内核尽力将新分配的线性区与紧邻的现有线性区合并
  - 如果两个相邻区的访问权限相匹配

增加或删除线性地址区的示意图

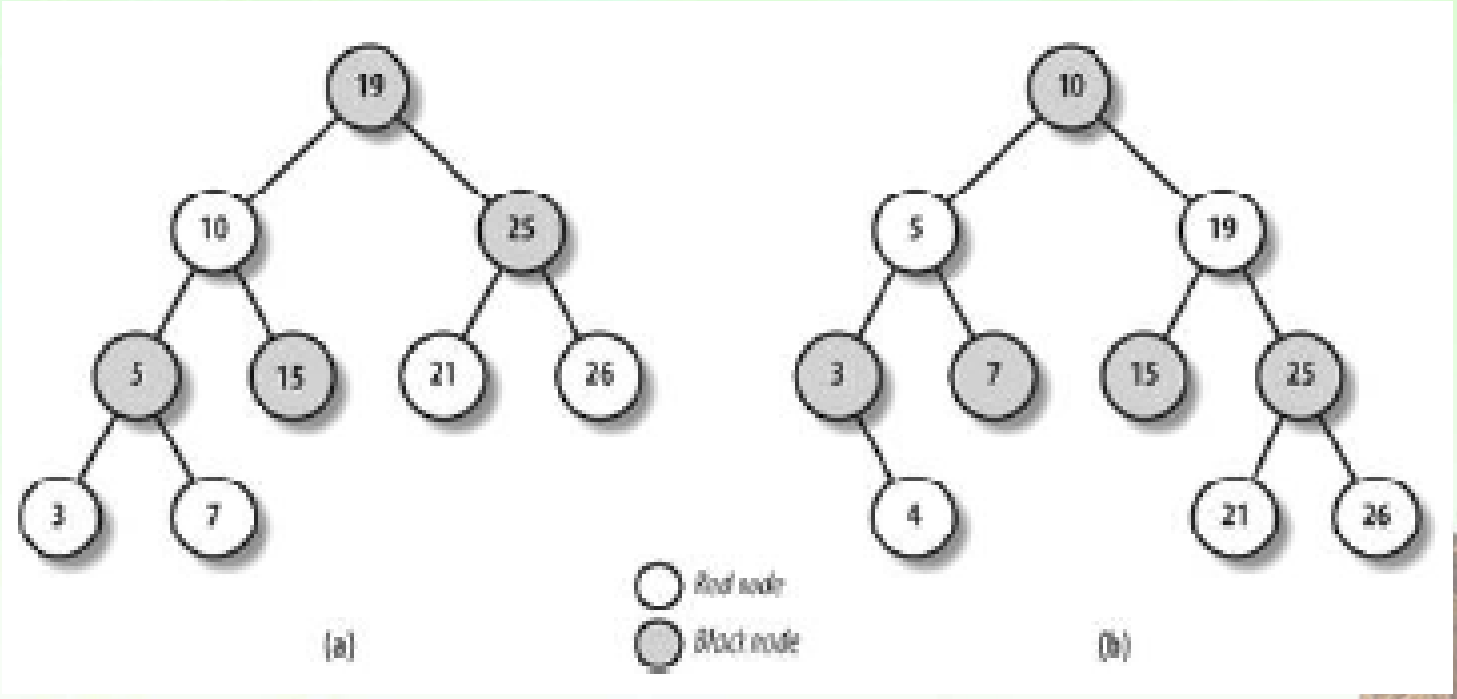


线性区的链表结构和红黑树结构

- 内存描述符中的**mmap**和**mm_rb**都可以访问线性区
  - 两者都指向同一个vm_area_struct结构，但链接方式不同
  - mmap指向的线性区链表用来遍历整个进程的地址空间
  - 红黑树用来定位一个给定的线性地址落在进程地址空间中的哪一个线性区中
    - ✓ 每个元素通常有两个孩子：左孩子和右孩子
    - ✓ 树中的元素的排序方式
      - 对每个接点N，N的左子树上的所有元素都派在N之前， N的右子树上的所有元素都派在N之后
    - ✓ 红黑树须满足的规则
      - 每个节点必须为黑或红
      - 树根必须为黑
      - 红节点的孩子必须为黑
      - 从一个节点到后代叶子节点的每个路径都包含相同数量的黑节点，统计黑节点个数时，空指针也算黑节点

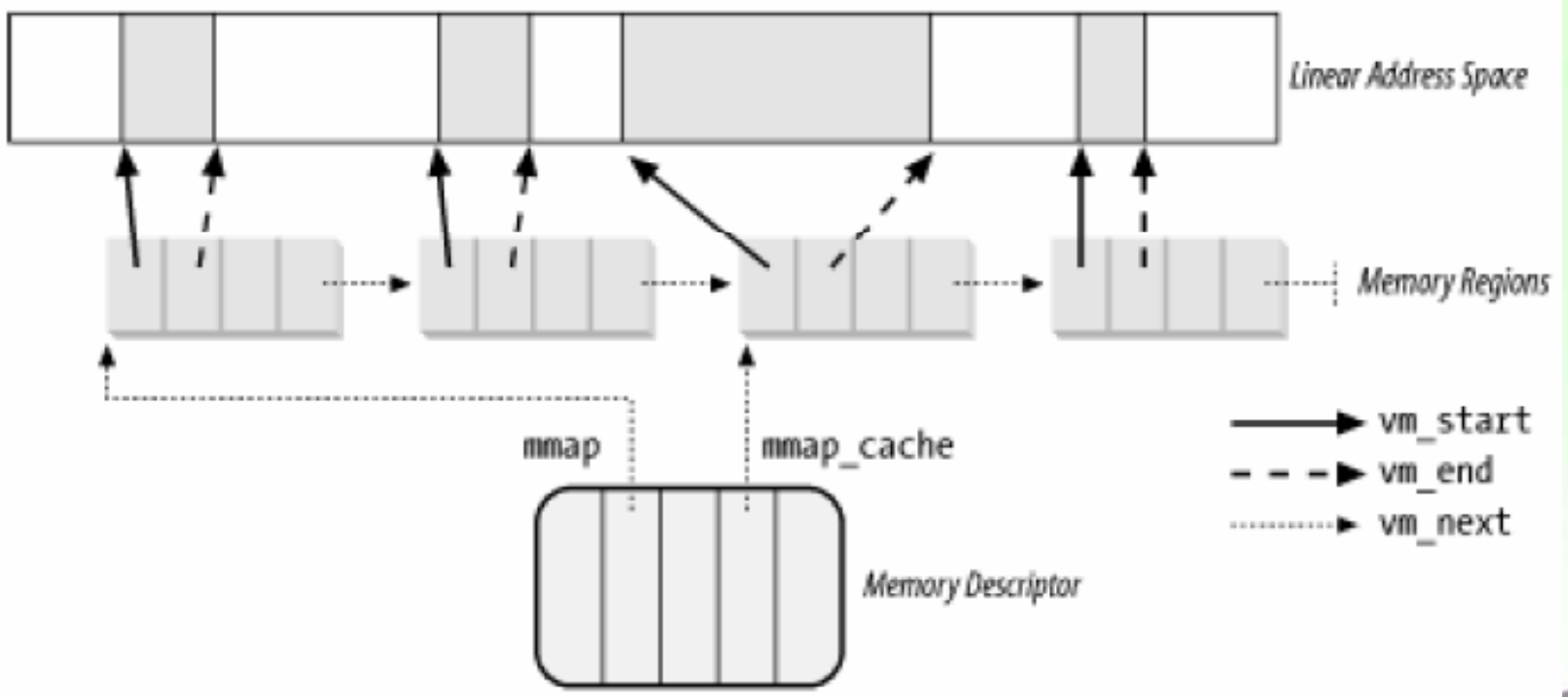
红黑树特点

- 树高度最多为2*long(n+1)



与进程地址空间相关的描述符

- **mmap_cache是进程最后一次引用线性区的描述符地址**



主要内容

- 背景知识
  - x86的分段机制
  - 物理存储管理
  - 进程虚拟存储管理
    - ✓ 进程的地址空间
    - ✓ 内存描述符
    - ✓ 线性区描述符
    - ✓ 线性区的处理
    - ✓ 虚存与实存的映射
- 实验内容
  - 统计系统和单个进程的缺页次数

线性区的处理

- 查找线性区
- 查找空闲进程地址区间
- 向内存描述符链表插入线性区
- 分配/释放线性地址区间

查找给定地址的最邻近线性区

- find_vma(struct mm_struct *mm, unsigned long addr)
  - 功能说明
    - 在指定地址空间搜索第一个vm_end大于addr的内存区域
    - 如果没有发现，则返回NULL
    - 否则返回匹配线性区的vm_area_struct结构体地址
  - 参数说明
    - mm：待搜索空间，进程内存描述符的地址
    - addr：线性地址
  - 工作流程
    - 首先查找mmap_cache
    - 如果没有，则搜索红黑树

find_vma()核心代码

```
vma = mm->mmap_cache;
if (vma && vma->vm_end > addr && vma->vm_start <= addr)
    return vma;
rb_node = mm->mm_rb.rb_node;
vma = NULL;
while (rb_node) {
    vma_tmp = rb_entry(rb_node, struct vm_area_struct, vm_rb);
    if (vma_tmp->vm_end > addr) {
        vma = vma_tmp;
        if (vma_tmp->vm_start <= addr)
            break;
        rb_node = rb_node->rb_left;
    } else
        rb_node = rb_node->rb_right;
}
if (vma)
    mm->mmap_cache = vma;
return vma;
```

查找与给定地址线性区相重叠的线性区

- **find_vma_intersection(struct mm_struct *mm, unsigned long start_addr, unsigned long end_addr)**
  - 功能说明
    - ✓只有在VMA的起始地址在给定的地址区间结束位置之前才将其返回
    - ✓如果VMA的其始位置大于指定地址范围的结束位置，则返回NULL
  - 参数说明
    - ✓mm：待搜索空间，进程内存描述符的地址
    - ✓start_addr：区间的首地址
    - ✓end_addr：区间的尾地址

find_vma_intersection()核心代码

```
static inline struct vm_area_struct *
find_vma_intersection(struct mm_struct * mm,
                      unsigned long start_addr,
                      unsigned long end_addr)
{
        struct vm_area_struct * vma;

        vma = find_vma(mm,start_addr);
        if (vma && end_addr <= vma->vm_start)
                vma = NULL;
        return vma;
}
```

查找一个空闲的地址区间

- **get_unmapped_area(unsigned long len, unsigned long addr)**
  - 功能说明
    - ✓搜索进程的地址空间，以找到一个可以使用的线性地址区间
    - ✓查找成功返回新区间的起始地址，否则返回错误码
  - 参数说明
    - ✓**len**：指定区间的长度
    - ✓**addr**：必须从哪个地址开始查找

get_unmapped_area()核心代码

```
if (len > TASK_SIZE)
    return -ENOMEM;
addr = (addr +0xfff) &0xfffff000;
if (addr && addr + len <= TASK_SIZE) {
    vma = find_vma(current->mm, addr);
    if (!vma || addr + len <= vma->vm_start)
        return addr;
}
start_addr = addr = mm->free_area_cache;
for (vma = find_vma(current->mm, addr); ; vma = vma->vm_next) {
    if (addr + len > TASK_SIZE) {
        if (start_addr == (TASK_SIZE/3 +0xfff)&0xfffff000)
            return -ENOMEM;
        start_addr = addr = (TASK_SIZE/3 +0xfff)&0xfffff000;
        vma = find_vma(current->mm, addr);
    }
    if (!vma || addr + len <= vma->vm_start) {
        mm->free_area_cache = addr + len;
        return addr;
    }
    addr = vma->vm_end;
}
```

向内存描述符链表插入一个线性区

- insert_vm_struct(struct mm_struct *mm, struct vm_area_struct *vmp)
  - 功能说明
    - ✓ 调用find_vma_preapre()在红黑树中查找vmp插入位置
    - ✓ 调用vma_link()依次执行以下操作
      - 在mm->mmap所指向的连表中插入线性区
      - 在红黑树mm->mm_rb插入线性区
      - 如果线性区匿名，将其插入相应的anon_vma数据结构作为头节点的链表中
      - 递增mm->map_count计数器
  - 参数说明
    - ✓ mm：指定进程描述符的地址
    - ✓ vmp：待插入的vm_area_struct对象的地址
      - 线性区的vm_start和vm_end必须已经初始化过

创建一个线性区间

- static inline unsigned long do_mmap(struct file *file, unsigned long addr, unsigned long len, unsigned long prot, unsigned long flag, unsigned long offset)
  - 参数说明
    - ✓ 根据file参数映射指定的文件中偏移量为offset，长度为len的一段内容
      - 如果file为NULL，offset为0，则为匿名映射
      - 如果指定了文件名和偏移量，则为文件映射
    - ✓ addr参数指明从何处开始查找一段可用的空闲线性地址区间
    - ✓ Prot参数指定这个区间所包含的页的存取权限
    - ✓ flags参数指定这个创建的线性区本身的一些标志
  - 说明
    - ✓ 如果创建的地址区间和已存在的地址区间相邻，且有相同的访问权限，则将区间合并

释放一个线性区间

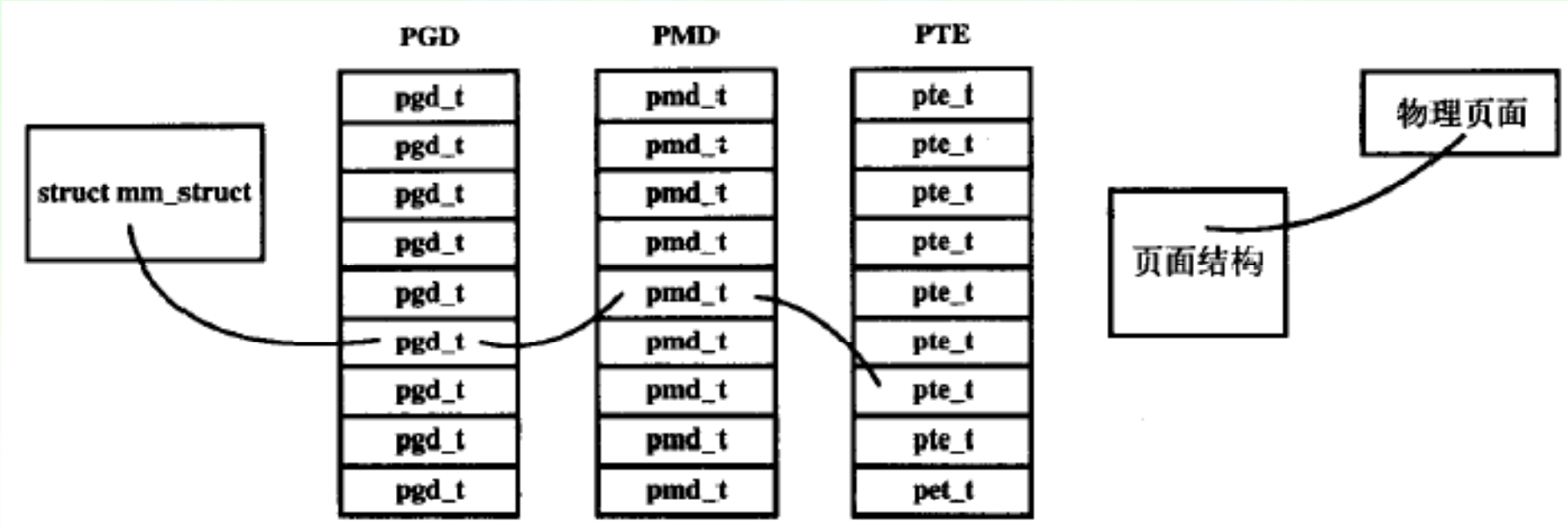
- int do_munmap(struct mm_struct *mm, unsigned long addr, size_t len)
  - 参数说明
  - mm: 指向当前进程的内存描述符
  - addr: 为线性区的起始地址
  - len: 指明要删除的区间大小

主要内容

- 背景知识
  - x86的分段机制
  - 物理存储管理
  - 进程虚拟存储管理
    - ✓ 进程的地址空间
    - ✓ 内存描述符
    - ✓ 线性区描述符
    - ✓ 线性区的处理
    - ✓ 虚存与实存的映射
- 实验内容
  - 统计系统和单个进程的缺页次数

虚存与实存的映射

- 通过查询页表完成
  - 将虚拟地址分段，每段作为一个索引指向页表
  - 页表则指向下一级别的页表或指向最终物理页面
- **Linux使用三级页表完成地址转换**



请页机制

- 实现虚存管理的重要手段
  - 进程运行时，CPU访问的是用户空间的虚地址
  - Linux仅把当前要使用的少量页面装入内存，需要时再通过请页机制将特定的页面调入内存
  - 当要访问的虚页不在内存时，产生一个页故障并报告故障原因

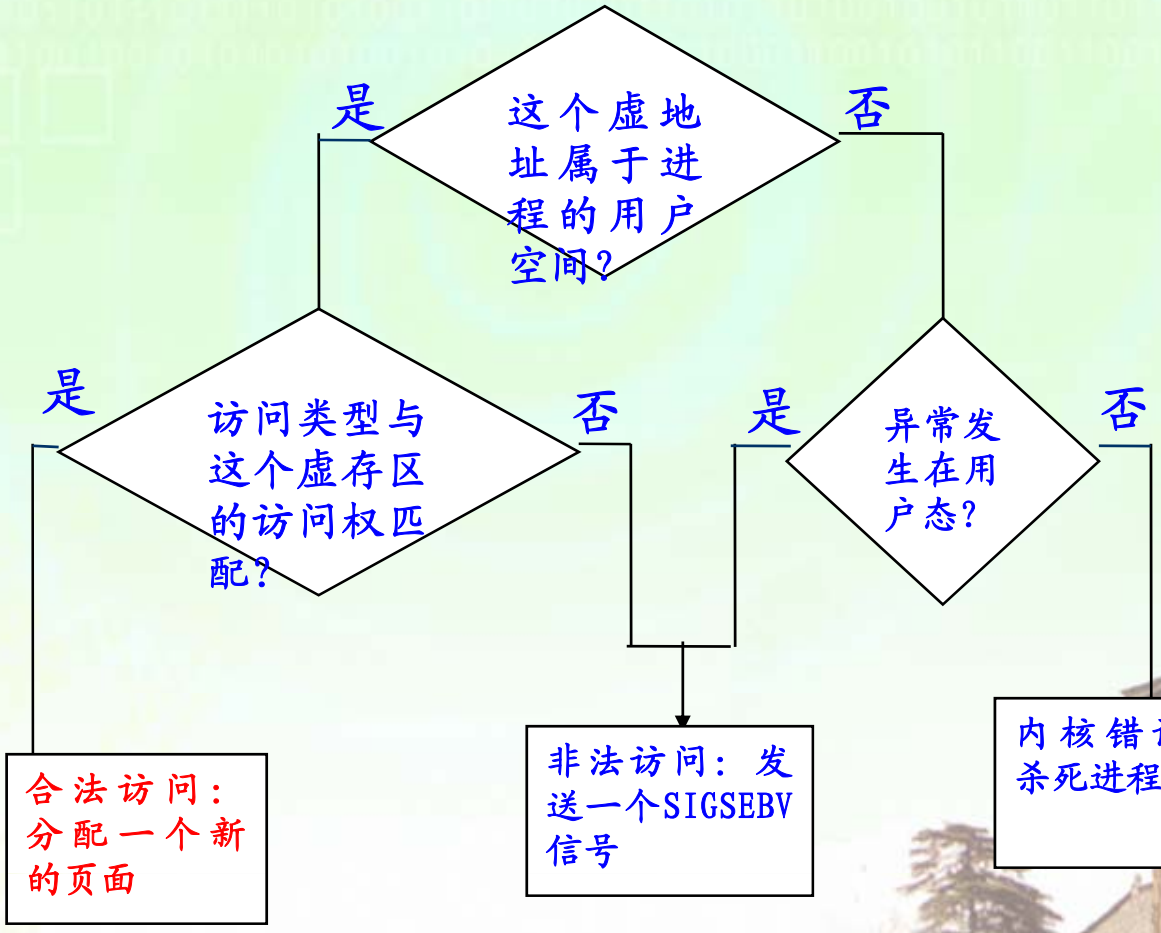
页故障原因

- 程序出现错误
  - 如要访问的虚地址在PAGE_OFFSET（3GB）之外，则该地址无效，Linux 将向进程发送一个信号并终止进程的运行
- 缺页异常
  - 虚地址有效，但其所对应的页当前不在物理内存中
  - 操作系统必须从磁盘或交换文件（此页被换出）中将其装入物理内存
- 保护错误
  - 要访问的虚地址被写保护
  - 操作系统必须判断
    - ✓ 如果是某个用户进程正在写当前进程的地址空间，则发送一个信号并终止进程的运行
    - ✓ 如果错误发生在一旧的共享页上时，则处理方法有所不同，也就是要对这一共享页进行复制，这就是曾经描述过的“写时复制”技术

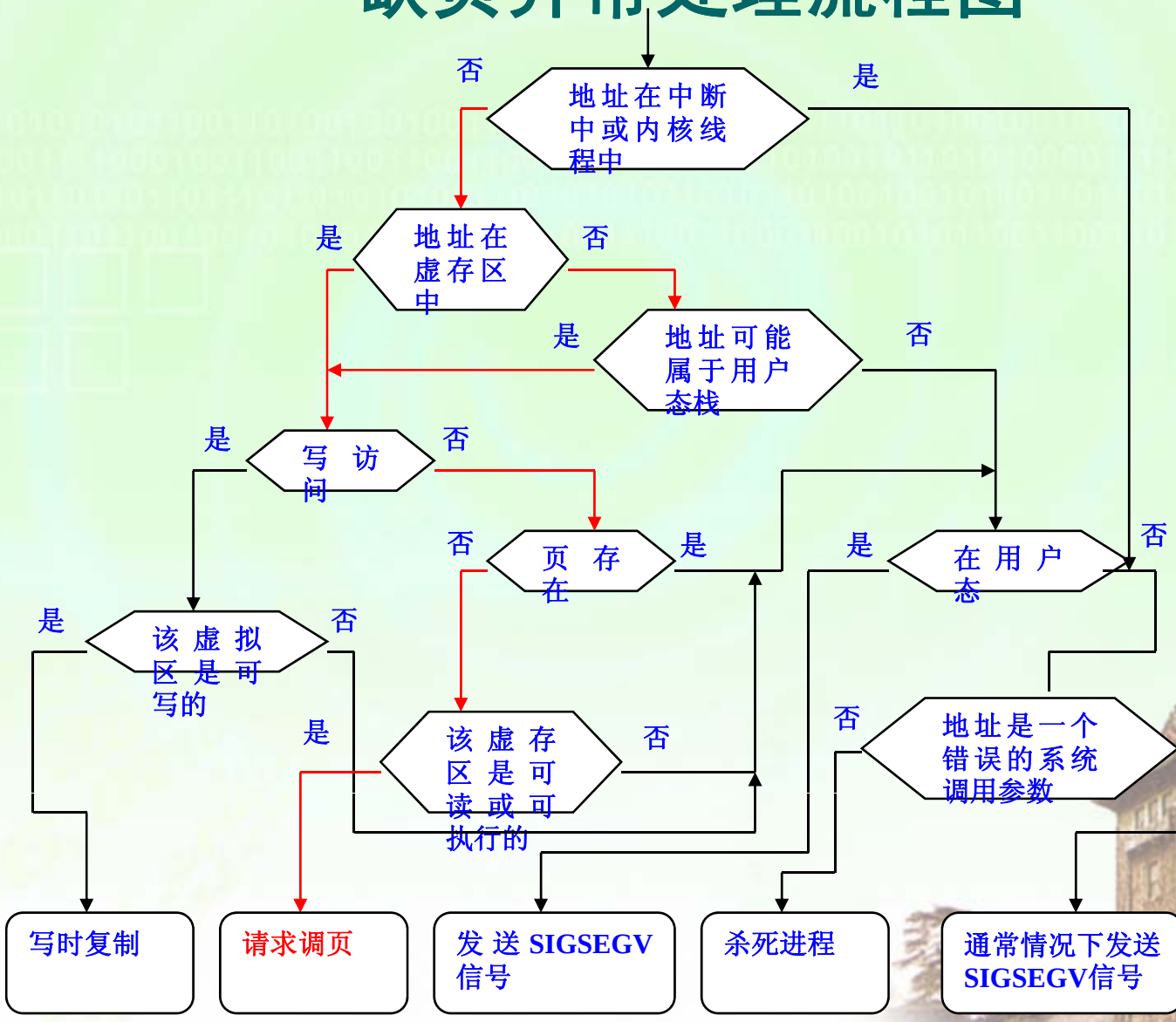
缺页异常

- i386中14号异常，由do_page_fault()完成
- 内核只是通过mmap()等调用分配一些线性地址空间给进程，并没有真正的把实际的物理页框分配给进程
  - 当进程试图访问这些分配给它的地址空间（如一段线性地址空间映射的是二进制代码）则进程被调度执行的时候会跳转到这个地址上去执行
  - 此时，并没有物理页框对应于这些线性地址，从而会引发一个缺页异常

缺页异常处理程序


这个虚地址属于进程的用户空间?
是
否
访问类型与这个虚存区的访问权匹配?
是
否
异常发生在用户态?
是
否
合法访问：分配一个新的页面
非法访问：发送一个SIGSEBV信号
内核错误：杀死进程

缺页异常处理流程图


地址在中断中或内核线程中
否
是
地址在虚存区中
是
否
地址可能属于用户态栈
是
否
写访问
是
否
页存在
否
是
在用户态
是
否
该虚拟区是可写的
是
否
该虚存区是可读或可执行的
是
否
地址是一个错误的系统调用参数
否
是
写时复制
请求调页
发送SIGSEGV信号
杀死进程
通常情况下发送SIGSEGV信号

请求调页

- 把页面的分配推迟到进程要访问的页不在物理内存时为止，由此引起一个缺页异常
  - 引入原因
    - ✓ 进程开始运行时并不访问其地址空间中的全部地址
    - ✓ 程序的局部性原理保证请求调页从总体上使系统有更大的吞吐量
  - 处理方法
    - ✓ 通过do_no_page()/do_anonymous_page()完成

主要内容

- 背景知识
  - x86的分段机制
  - 物理存储管理
  - 进程虚拟存储管理
  - slab分配器
- 实验内容
  - 统计系统和单个进程的缺页次数

统计系统和单个进程的缺页次数

❖实验说明

- 修改系统的缺页异常处理程序，使之能够记录缺页次数
  - ✓系统缺页次数
  - ✓单个进程缺页次数，并提供新的系统调用供用户查询缺页次数

统计系统和单个进程的缺页次数

❖解决方案：统计方法

- 当每次发生缺页时，缺页中断服务内核函数do_page_fault( )都要被调用，所以可以认为执行该函数的次数就是系统发生缺页的次数
- 需要修改do_page_fault( )，使它在每次被调用的时候对一个计数器进行自增，该计数器的值便是整个系统发生缺页的次数
- 为了统计进程发生缺页的次数，需要在进程的进程控制块中添加一个成员用于记录进程发生缺页的次数
- 当do_page_falut( )被调用时，函数通过current宏获得当前进程的进程控制块，并将其中用于记录缺页次数的成员进行自增

统计系统和单个进程的缺页次数

❖解决方案：记录全局缺页次数

- 先需要添加一个全局变量global_pf作为计数变量，将该变量申明在include/Linux/mm.h文件中
  - ✓extern unsigned long volatile global_pf;
- 在arch/i386/mm/fault.c中定义该变量
  - ✓unsigned long volatile global_pf;
- 修改arch/i386/mm/fault.c中的do_page_fault( )，在该函数的开始时将global_pf自增:

```
fastcall void __kprobes do_page_fault(struct pt_regs
  *regs, unsigned long error_code) {
  global_pf++;
  ……
}
```

统计系统和单个进程的缺页次数

❖解决方案：记录进程缺页次数

- 首先要在进程的进程控制块中添加一个成员pf
- 进程控制块task_struct定义在include/Linux/Sched.h中
  - ✓记录进程缺页次数与记录系统缺页次数类似
- 修改arch/i386/mm/fault.c中的do_page_fault( )函数，在该函数开始时将当前进程的pf自增:

```
fastcall void __kprobes do_page_fault(struct pt_regs
  *regs, unsigned long error_code) {
  global_pf++;
  current->pf++;
  ……
}
```

统计系统和单个进程的缺页次数

- 解决方案：统计进程缺页次数与系统缺页次数区别
  - 在进程创建时需要将进程控制块中的pf设置为0
  - 在Linux中，进程创建是采用fork( )操作，在进程创建过程中，子进程会将父进程的进程控制块拷贝一份
  - 实现该拷贝过程的函数是kernel\fork.c文件中的dup_task_struct( )函数，修改该函数将子进程pf设置成0

```
static struct task_struct *dup_task_struct(struct
  task_struct *orig) {
  ……
  *tsk = *orig;
  tsk->pf = 0;
  ……
}
```

统计系统和单个进程的缺页次数

- 解决方案：添加系统调用
  - 需要添加两个系统调用
    - ✓ sys_global_pf()
    - ✓ sys_pf()
  - 这两个系统调用只要将global_pf和当前进程控制块中的pf值返回即可

```
int sys_global_pf( ) {
    ……
    return global_pf;
}
int sys_pf( ) {
    ……
    return current->pf;
}
```

第16章 存储管理