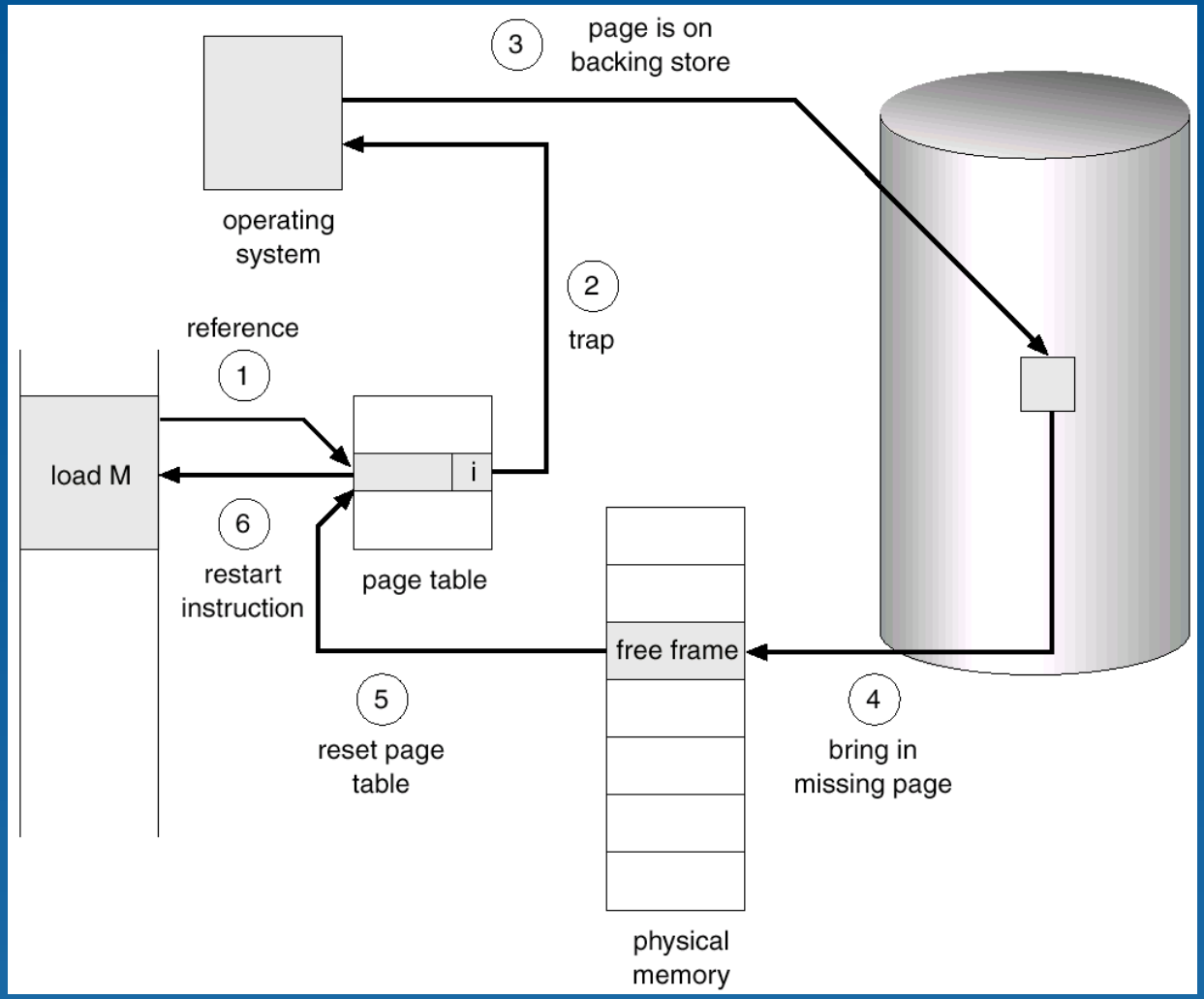

# Linux 存储管理概述

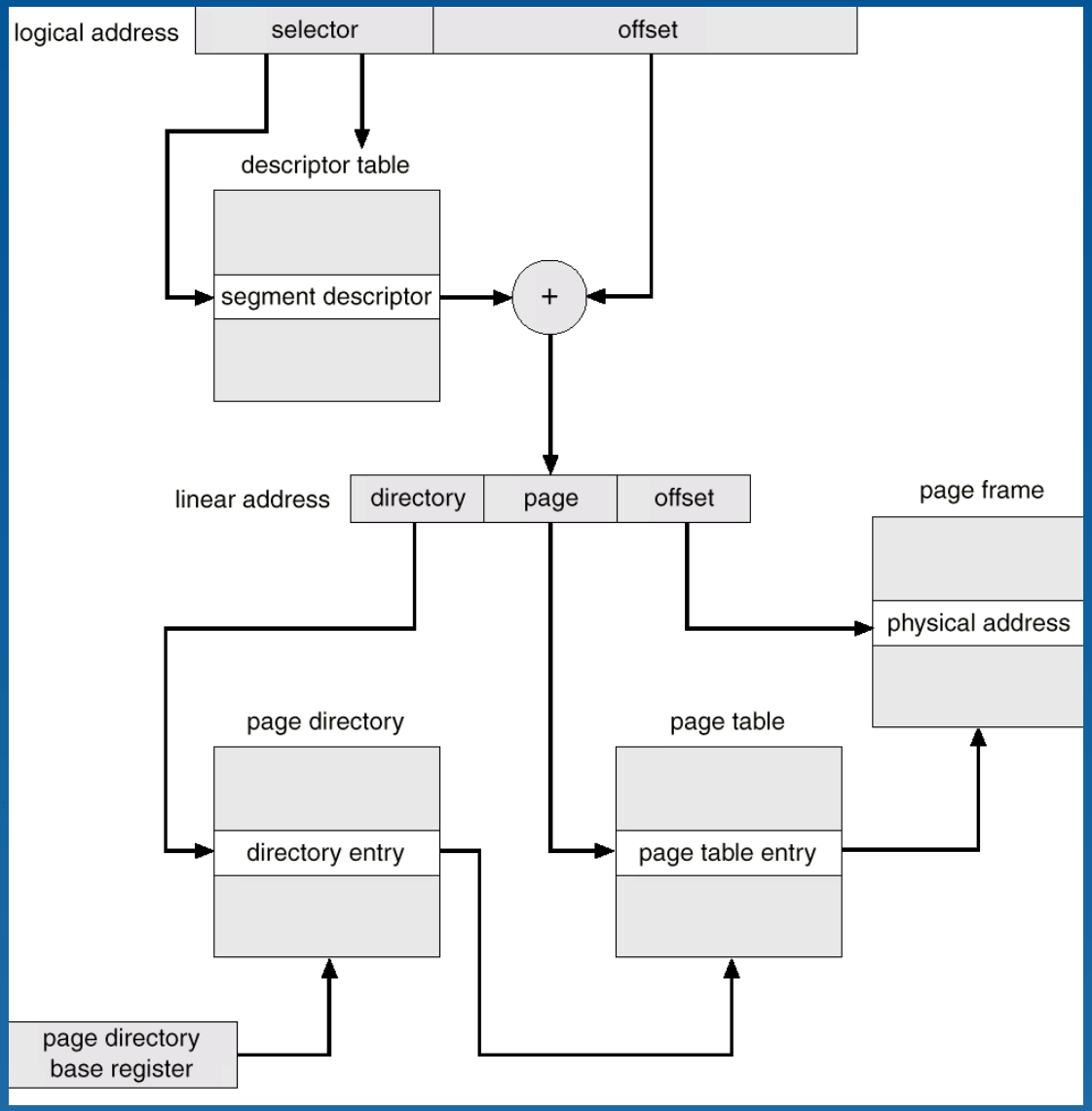
# Linux 存储管理要点

- i386 MMU
- 虚拟存储组织结构 (3 个途径 )
- 存储保护
- 物理空间管理
- 空闲物理空间
- 内核态的物理内存，分配 & 释放
- 内核态的虚拟地址空间，分配 & 释放
- 内核态交换进程，页面换出操作
- 缺页及其响应，页面换入操作
- Cache

# 示意图：缺页及其处理


3
page is on
backing store
operating
system
2
trap
reference
1
load M
i
6
restart
instruction
page table
free frame
5
reset page
table
4
bring in
missing page
physical
memory

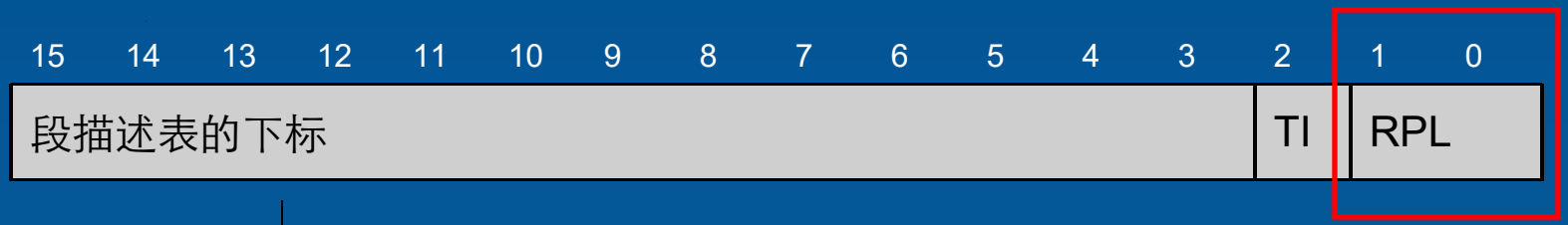
# 示意图： i386 MMU


logical address
selector
offset
descriptor table
segment descriptor
+
linear address
directory
page
offset
page frame
physical address
page directory
directory entry
page table
page table entry
page directory
base register

# i386 的选择字 (The Selector)

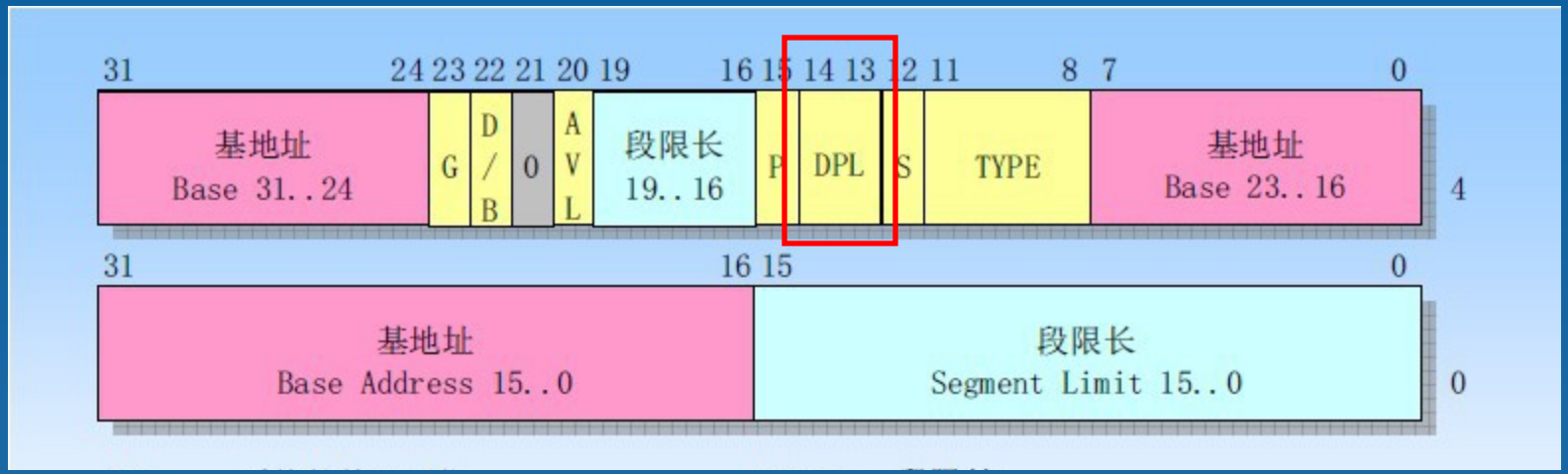

- TI=0， Selector 指向段表 GDT 的一个段表项
- TI=1， Selector 指向段表 LDT 的一个段表项
- RPL，此次内存访问的申请者所拥有的特权级 (Privilege level)
  - 最高级是 0
  - 最低级是 3
- INDEX，指向段表 GDT/LDT 的下标值

# 段式机制




GDT / LDT

RPL 优先级高于 DPL 才允许访问



# 段表项的定义

| Location | Description |
|---|---|
| bit 15 – bit 00 | Segment limit, bits 15 : 00 |
| bit 31 – bit 16 | Segment addr., bits 15 : 00 |
| bit 39 – bit 32 | Segment addr., bits 23 : 16 |
| bit 47 – bit 40 | access rights |
| bit 51 – bit 48 | Segment limit, bits 19 : 16 |
| bit 52 | Defined by user |
| bit 53 | 0 (reserved) |
| bit 54 | D |
| bit 55 | G |
| bit 63 – bit 56 | Segment addr., bits 31 : 24 |

# G, granularity

- G=0，以“字节”为最小单位
- G=1，以“ 4K 字节”为最小单位

# Linux 不得不使用 GDT 的极小部分

| | | | | |
|---|---|---|---|---|
| 0 | NULL descriptor | | | |
| 1 | Not used | | | |
| 2 | Code in Kernel Mode | Virtual addr. Starts at 0XC0000000 | 1GB | Privilege level 0 |
| 3 | Data in Kernel Mode | Virtual addr. Starts at 0XC0000000 | 1GB | Privilege level 0 |
| 4 | Code in User Mode | Virtual addr. Starts at 0X00000000 | 3GB | Privilege level 3 |
| 5 | Data in User Mode | Virtual addr. Starts at 0X00000000 | 3GB | Privilege level 3 |
| 6 | Not used | | | |
| 7 | Not used | | | |
| 2i+6 | LDT of Process i | | | |
| 2i+7 | TSS of Process i | | | |

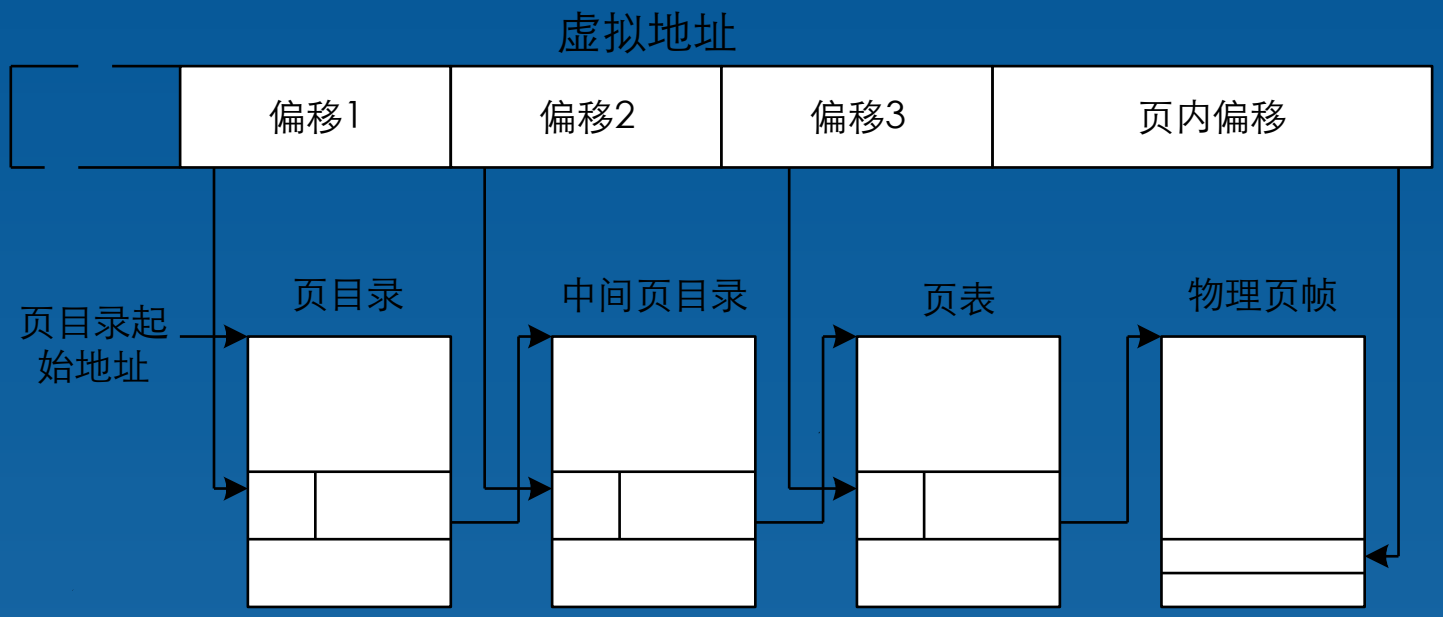
# Linux 不得不使用 LDT 的极小部分

| | | | | |
|---|---|---|---|---|
| 0 | NULL descriptor | | | |
| 1 | Code in User Mode | Virtual addr. Starts at 0X00000000 | 3GB | Privilege level 3 |
| 2 | Data in User Mode | Virtual addr. Starts at 0X00000000 | 3GB | Privilege level 3 |

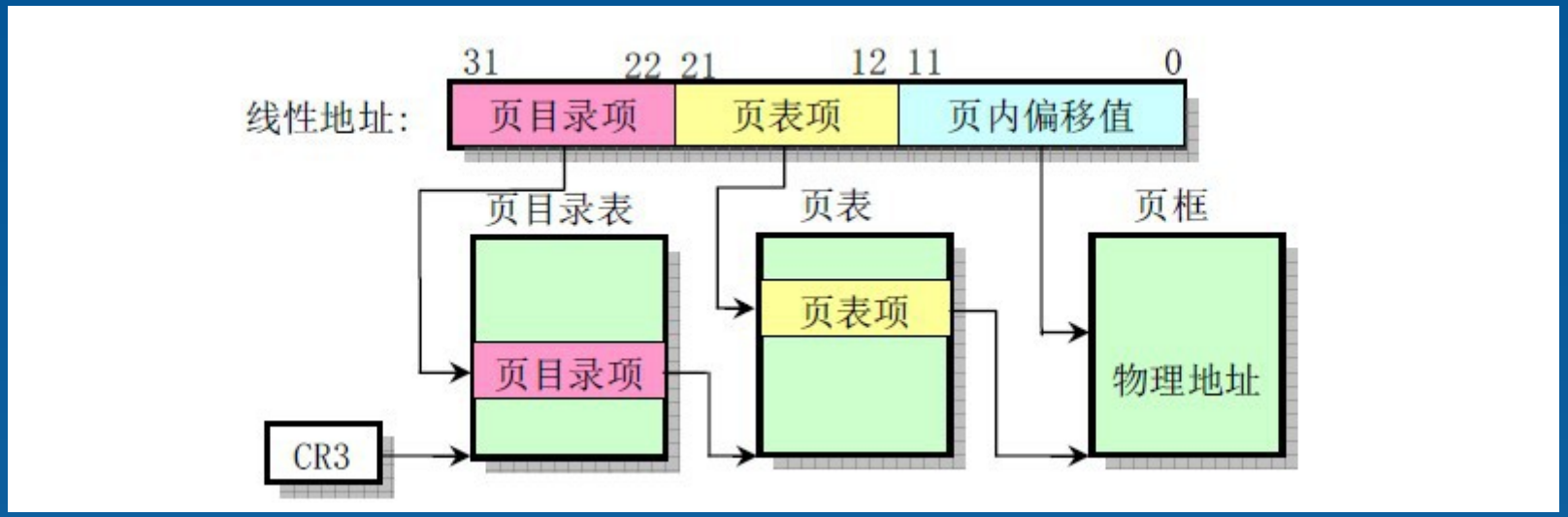
# 控制寄存器

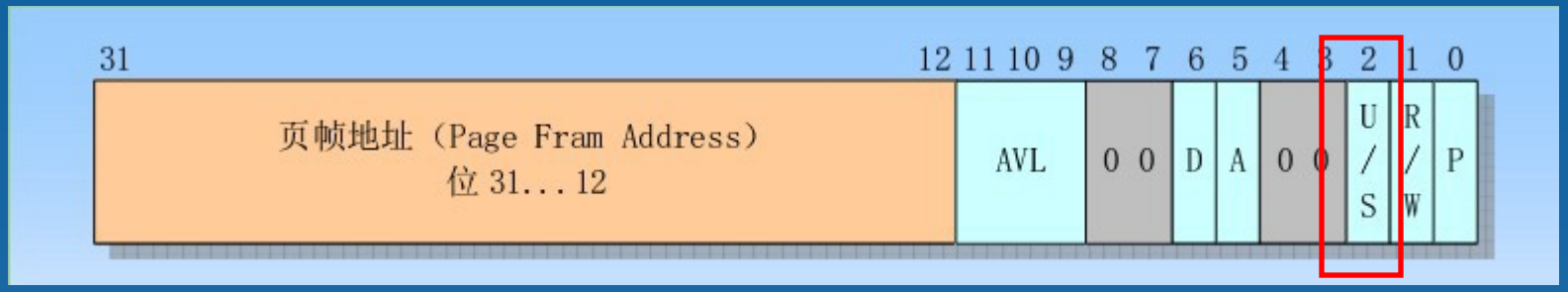
- CR3 指示页目录表的起始地址
- CR0 寄存器的 PE 位 ( 位 0) 用于控制段机制。 PE=1 则处理器工作于保护模式下； PE=0 则处理器工作于实模式下，等同于 8086
- CR0 的 PG 位 ( 位 31) 用于控制分页机制。 PG=1 则启用分页机制， 32 位线性地址通过页表转换为物理地址； PG=0 则禁止分页机制， 32 位线性地址直接寻址物理地址
- 当 PE=1 且 PG=0 时，处理器工作于保护模式下，但禁止分页机制。此时没有内存和磁盘之间的页面交换，也就不存在虚拟内存
- 当 PE=1 且 PG=1 时，处理器工作于保护模式下，但启用分页机制。此时有内存和磁盘之间的页面交换，磁盘起到虚拟内存的作用
- CR2 指示引起缺页中断的地址

# Linux 分页管理，适应 i386 架构

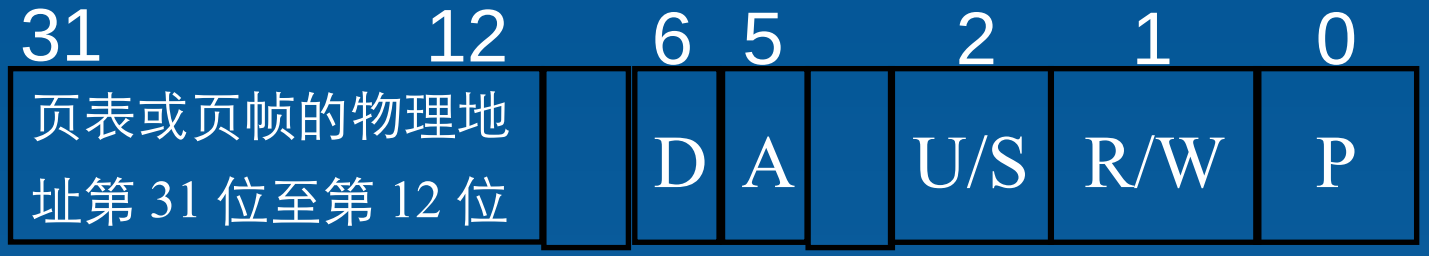


# 页式机制


31
22 21
12 11
0
线性地址:
页目录项
页表项
页内偏移值
页目录表
页表
页框
页表项
页目录项
物理地址
CR3


31
12 11 10 9 8 7 6 5 4 3 2 1 0
页帧地址（Page Fram Address）
位 31...12
AVL
0 0
D
A
0 0
U / S
R / W
P

# 页目录项和页表项




P=1 则地址转换有效； P=0 则地址转换无效

R/W=1 则该页可写，可读，且可执行；
R/W=0 则该页可读，可执行，但不可写

U/S=1 则该页可在任何特权级下访问；
U/S=0 则该页只能在特权级 0 、 1 和 2 下访问

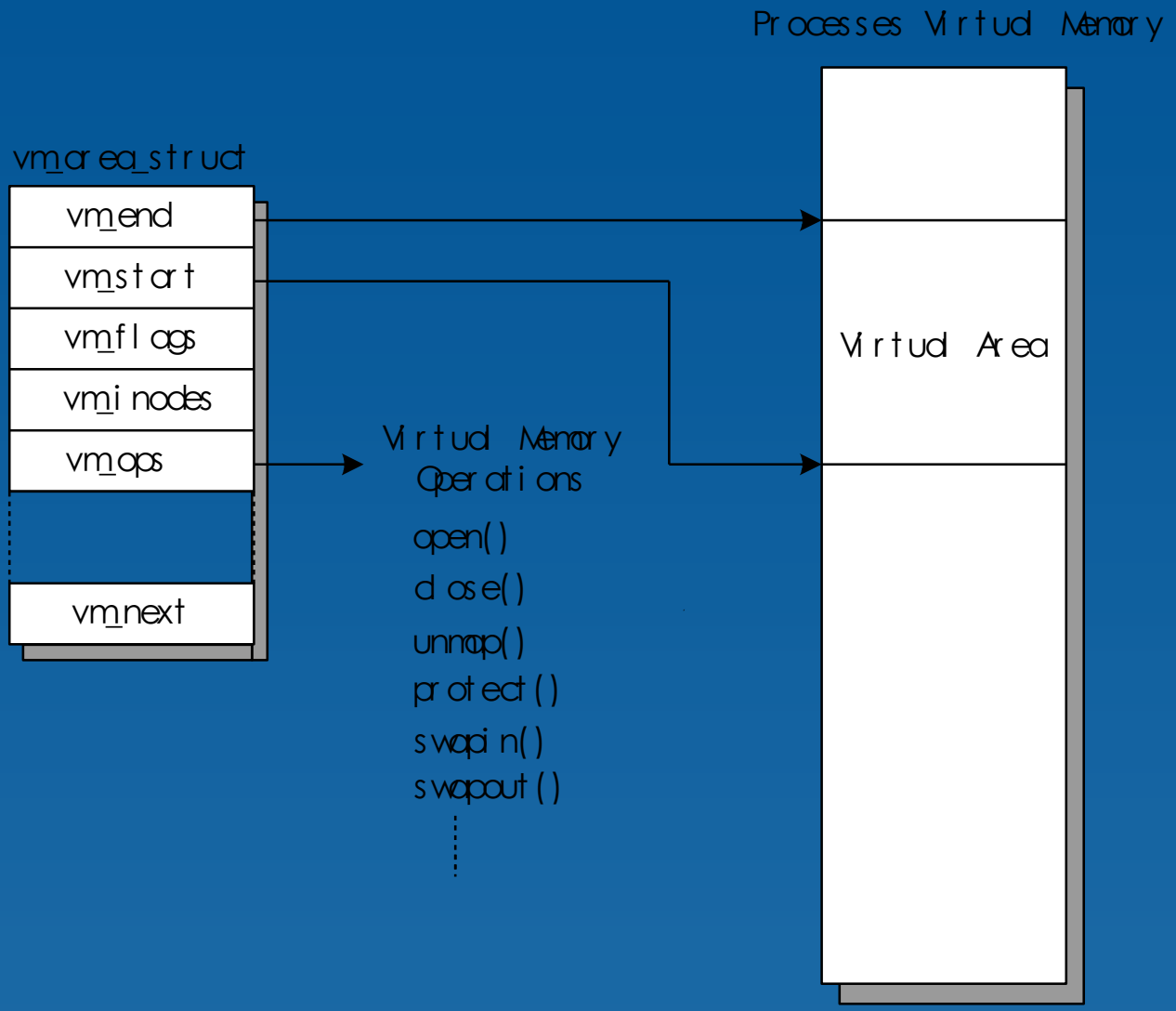
A： 访问位

D： 已写标志位

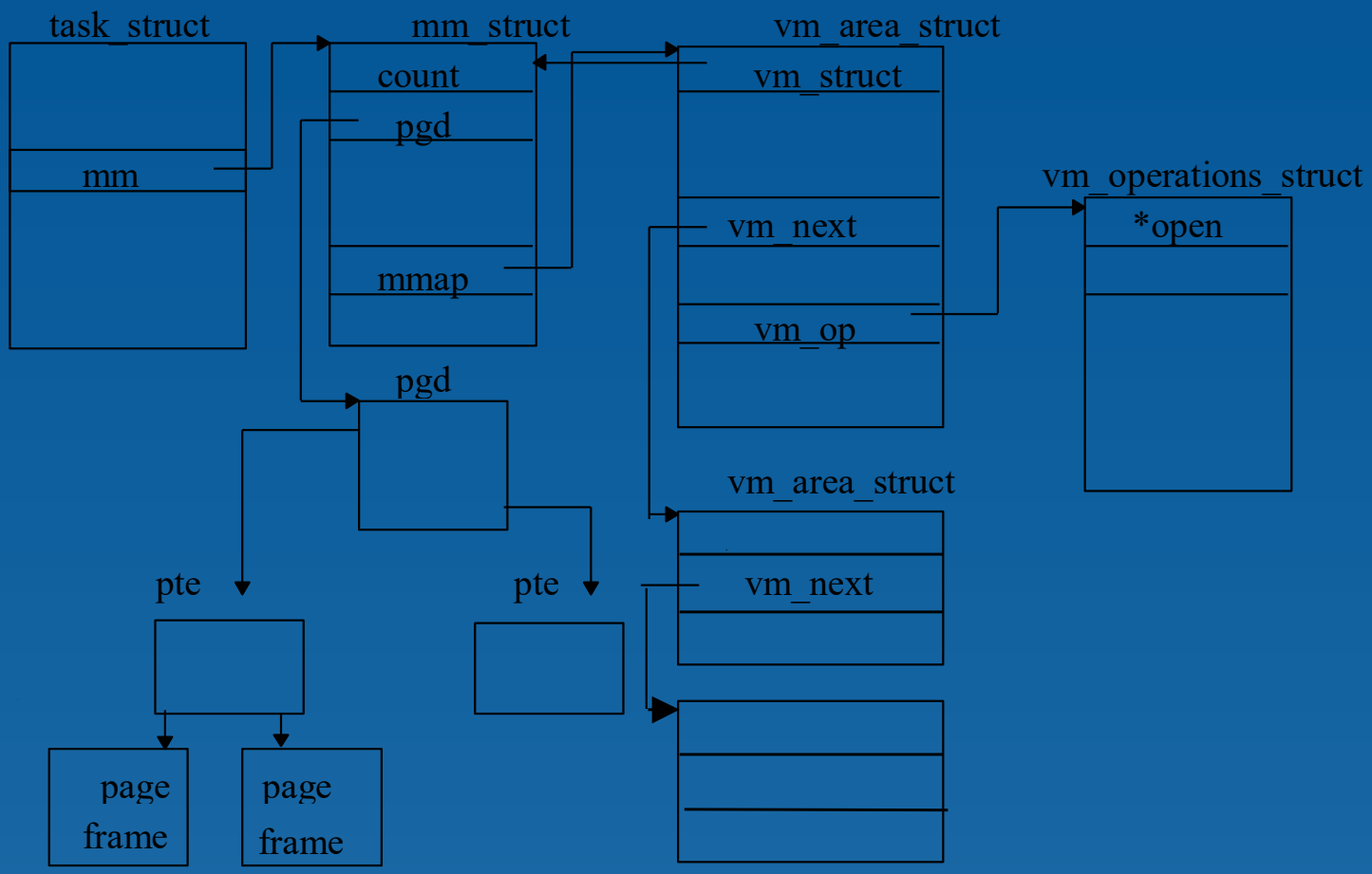
# PCB 对存储空间的管理

```
struct mm_struct {
  int count;
  pgd_t * pgd; /*  进程页目录的起始地址  */
  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;     /*  指向 vma 双向链表的指针  */
  struct vm_area_struct * mmap_avl; /*  指向 vma AVL 树的指针  */
  struct semaphore mmap_sem;
}
```

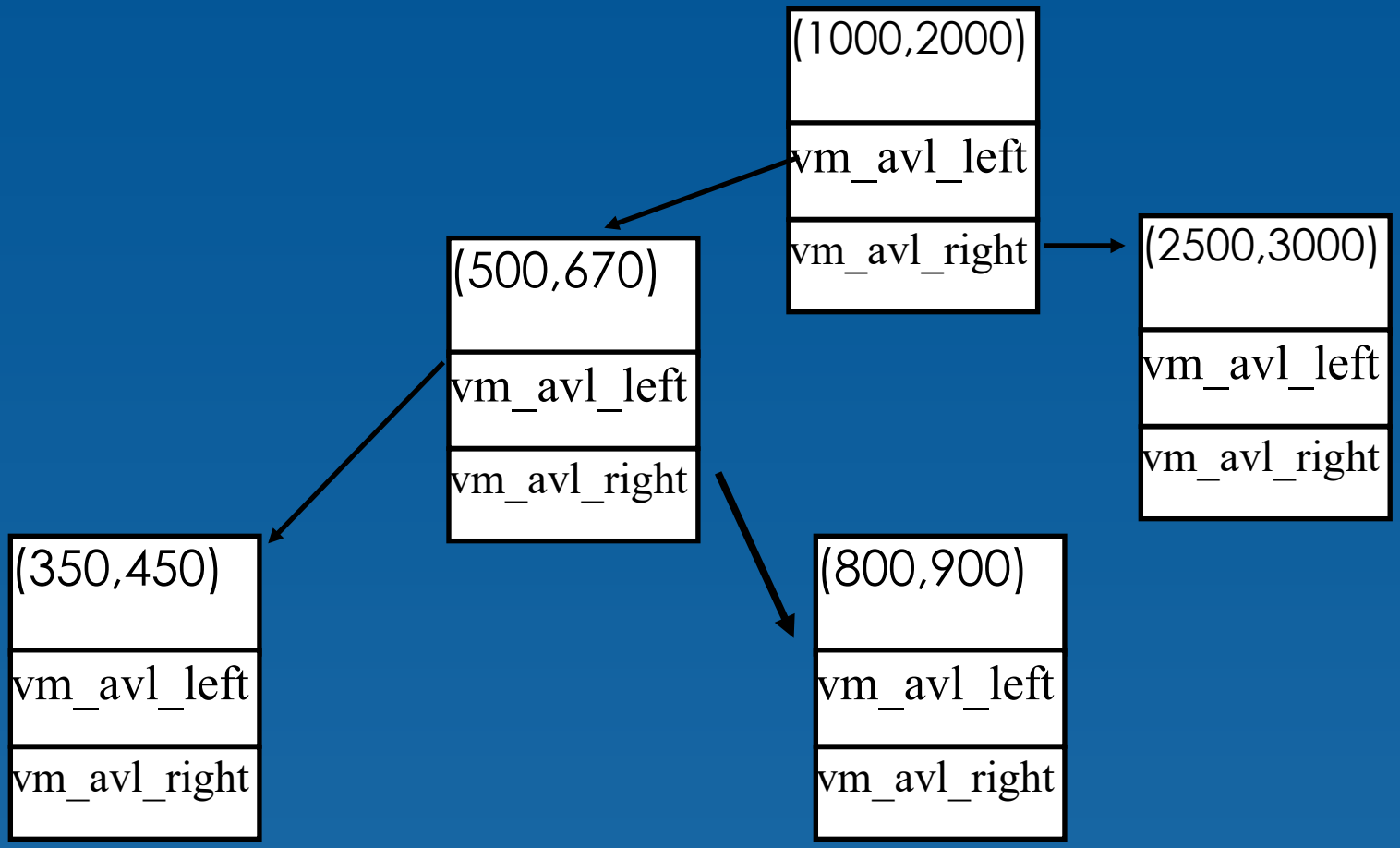
# 虚存段 vma


Processes Virtual Memory
vm_area_struct
vm_end
vm_start
vm_flags
vm_inodes
vm_ops
vm_next
Virtual Area
Virtual Memory Operations
open()
close()
unmap()
protect()
swapin()
swapout()

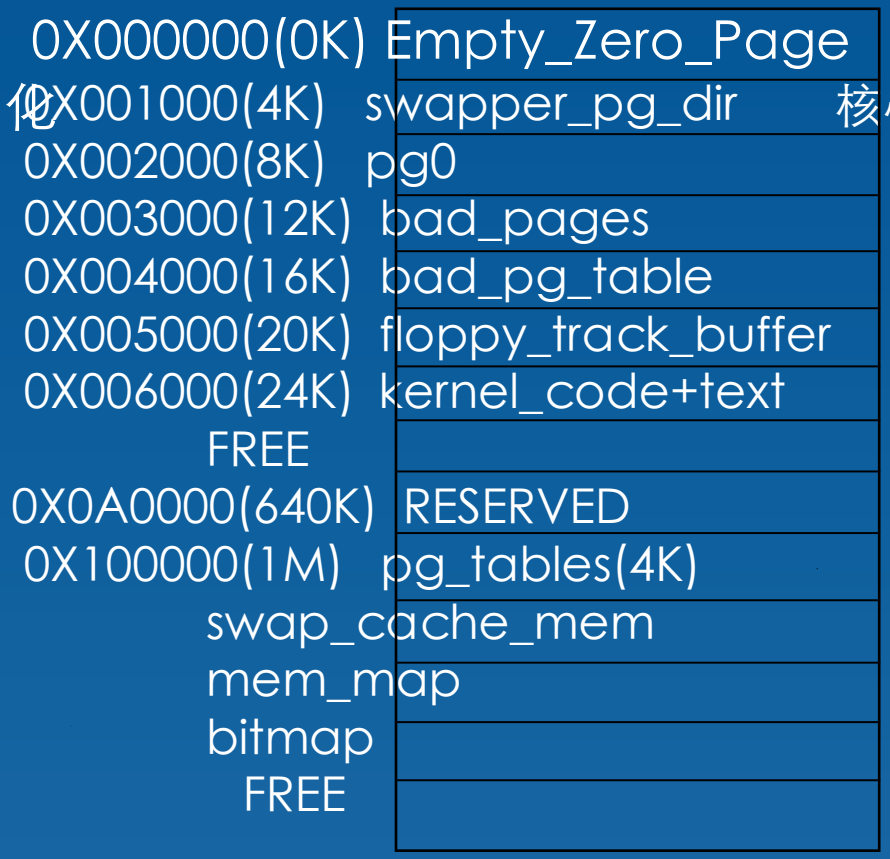
# 一个进程的虚拟空间


task_struct
mm
mm_struct
count
pgd
mmap
vm_area_struct
vm_struct
vm_next
vm_op
vm_operations_struct
*open
pgd
vm_area_struct
vm_next
pte
pte
page
frame
page
frame

# AVL 树（Adelson-Velskii and Landis）



# 初始化后物理存储分布




由 mem_init 初始化

核心态访问空间的页目录

# 物理页面

```
typedef struct page {
    struct page *next, *prev; /* 由于搜索算法的约定，这两项必须首先定义 */
    struct inode *inode;  /* 若该页帧的内容是文件，则 inode 和 offset
    unsigned long offset; /* 指出文件的 inode 和偏移位置 */
    struct page *next_hash;
    atomic_t count; /* 访问此页帧的进程记数，大于 1 表示由多个进程共享 */
    unsigned flags;   /* atomic flags, some possibly updated asynchronously */
    unsigned dirty:16, /* 页帧修改标志 */
             age:8;    /* 页帧的年龄，越小越先换出 */
    struct wait_queue *wait;
    struct page *prev_hash;
    struct buffer_head * buffers; /* 若该页帧作为缓冲区，则指示地址 */
    unsigned long swap_unlock_entry;
    unsigned long map_nr;       /* 页帧在 mem_map 表中的下标，
                                   page->map_nr == page - mem_map */
} mem_map_t;

mem_map_t * mem_map = NULL; /* 页帧描述表的首地址 */
```

# 空闲物理内存管理

- bitmap 表
- 在物理内存低端，紧跟 mem_map 表的 bitmap 表以位示图方式记录了所有物理内存的空闲状况。与 mem_map 一样， bitmap 在系统初始化时由 free_area_init() 函数创建 (mm/page_alloc.c)
- 与一般性位图不同， bitmap 表分割成 NR_MEM_LISTS 组 ( 缺省值 6) 。
- buddy 算法

  buddy 算法分配空闲块，由 _get_free_pages() 和 free_pages() 函数执行。 change_bit() 函数根据 bitmap 的对应组，判断回收块的前后邻居是否也为空 。

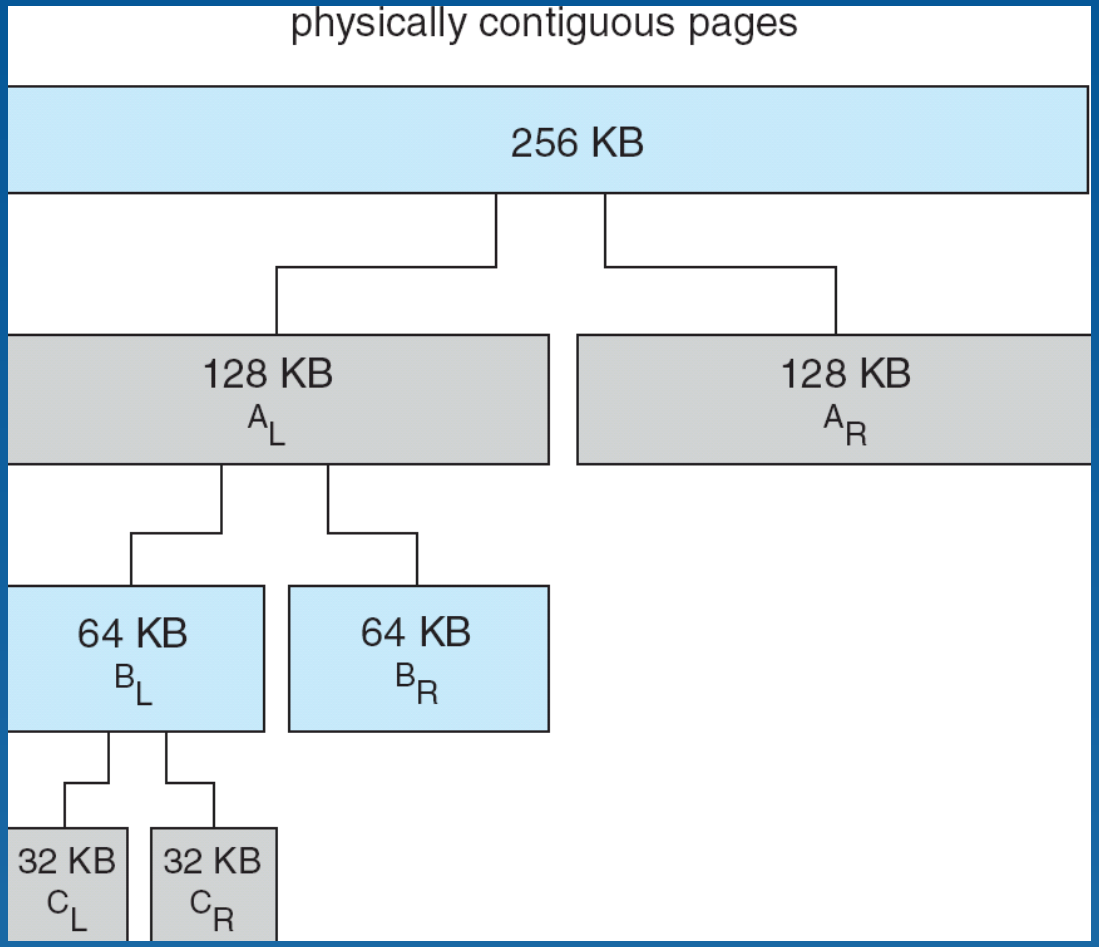
# 伙伴 (Buddy) 算法

- 以若干种固定长度的连续块分配、管理物理内存
- 固定长度取 2 的次幂（**power-of-2 allocator)**
  - 响应内存请求时，总是返回长度为“ 2 的次幂”的连续块
  - 连续块首地址以“ 2 的次幂”对齐，末地址与另一个“ 2 的次幂”对齐

# 伙伴 (Buddy) 算法

- 找到一块符合申请长度的连续块时，先不马上分出去
- 把数据块一分为二 ( 还是 “ 2 的次幂” )
- 判断分割后的数据块是否仍然满足申请长度
  - 如果满足，继续“一分为二”
  - 如果不满足，那么把分割前的连续块分配出去

# 示例：伙伴算法

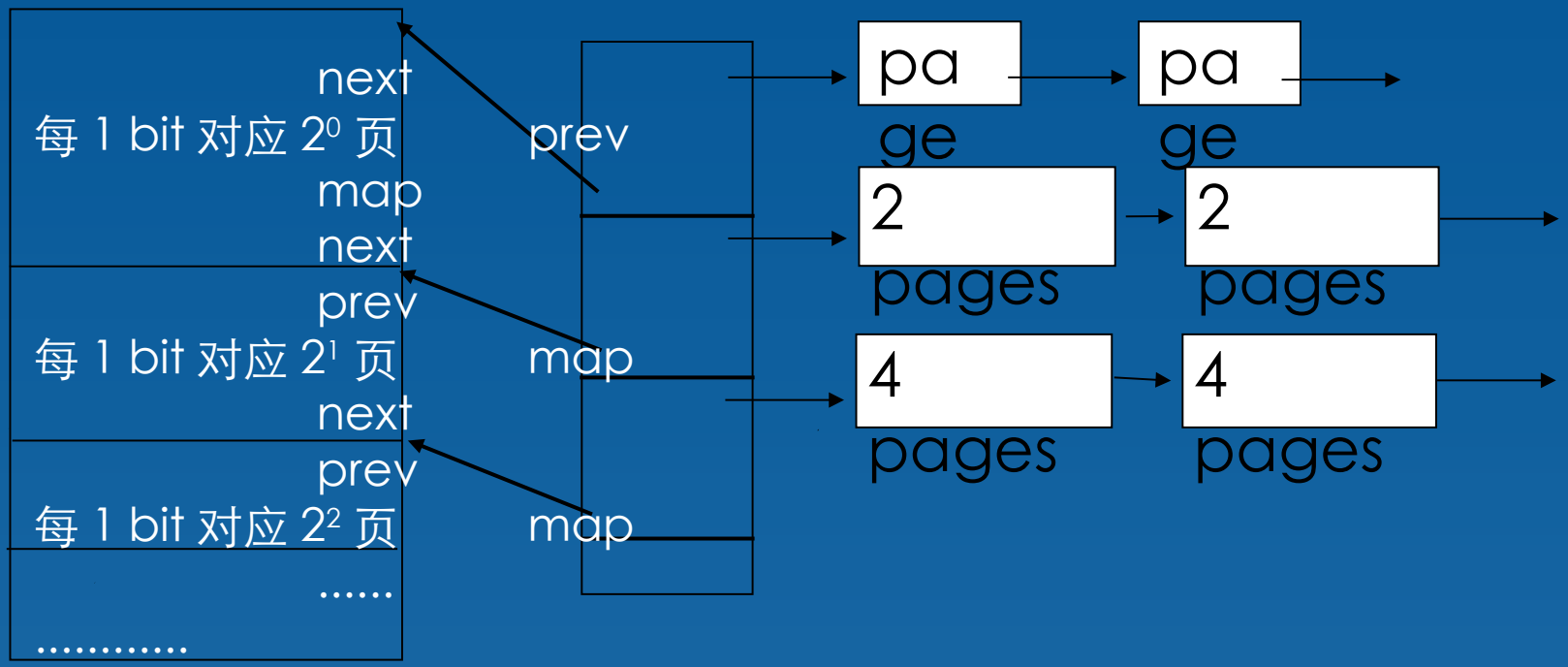

physically contiguous pages
256 KB
128 KB
$A_L$
128 KB
$A_R$
64 KB
$B_L$
64 KB
$B_R$
32 KB
$C_L$
32 KB
$C_R$

# 空闲物理内存管理

- LINUX 用 free_area 数组记录空闲的物理页帧

```
struct free_area_struct {
    struct page *next;
    /*  此结构的 next,prev 指针与 struct page 匹配  */
    struct page *prev;
    unsigned int * map; /*  指向 bitmap */
};
static struct free_area_struct free_area[NR_MEM_LISTS];
```

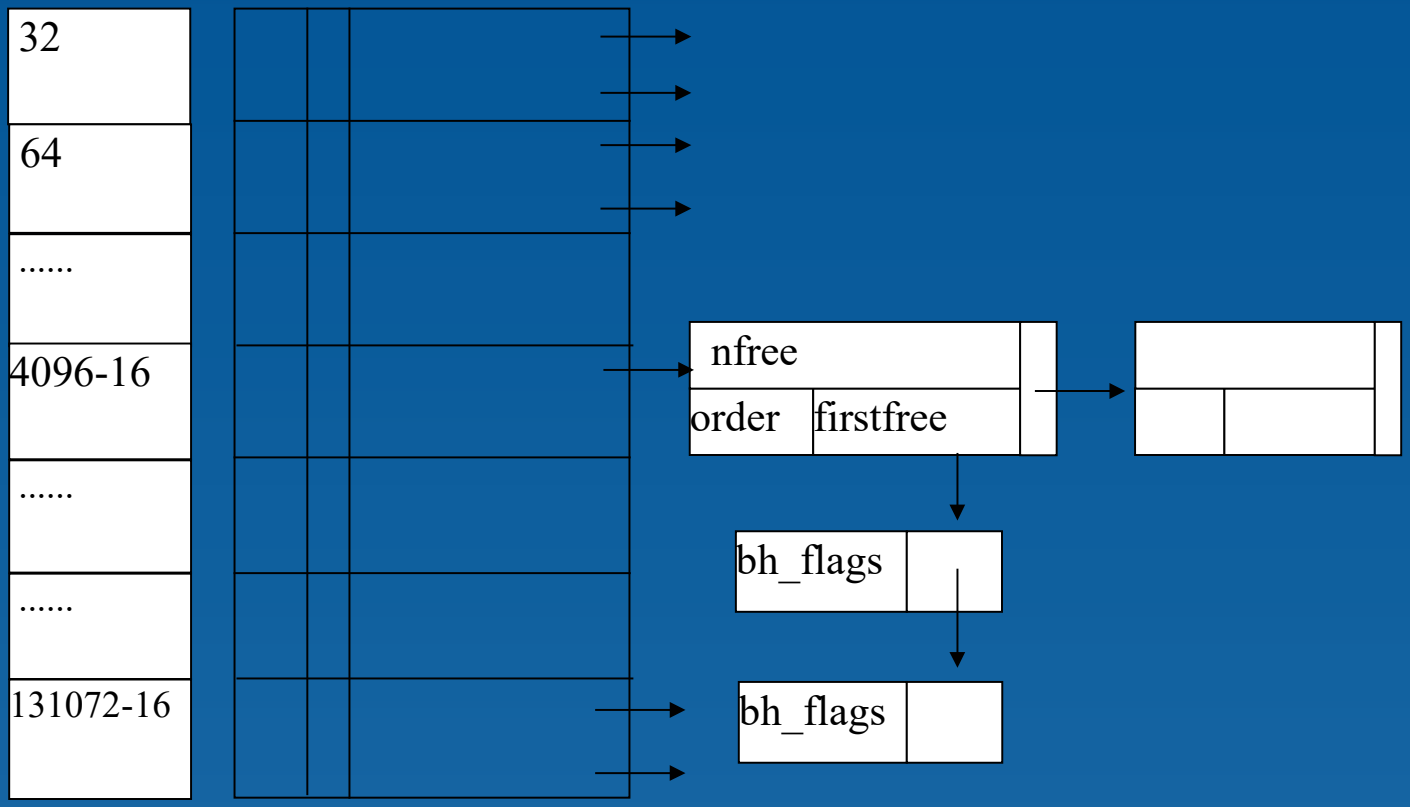
# 空闲物理内存管理


next
每 1 bit 对应 2^0 页
prev
map
next
prev
每 1 bit 对应 2^1 页
map
next
prev
每 1 bit 对应 2^2 页
map
.....
.....
page
page
2
pages
2
pages
4
pages
4
pages

# 内核态实存的申请与释放

- 早期版本的方法
  - kmalloc() 和 kfree()
  - kmalloc_cache
- 流行的方法： SLAB
- SLAB 可供用户使用的函数有
  - kmem_cache_create(), kmem_cache_alloc(), kmalloc(), kmem_cache_free,
  - kfree()/kfree_s(), kmem_cache_shrink(), kmem_cache_reap()

# kmalloc() / kfree()




blocksize 表

sizes 表

# Slab 算法

- **如果内核数据结构长度等于物理页帧长度**
  - 没有碎片
  - 快速响应内核内存请求
- **没有这么幸运，往往内核数据结构长度不等于物理页帧长度。仍然能够**
  - 没有碎片？
  - 快速响应内核内存请求？

# Slab 算法

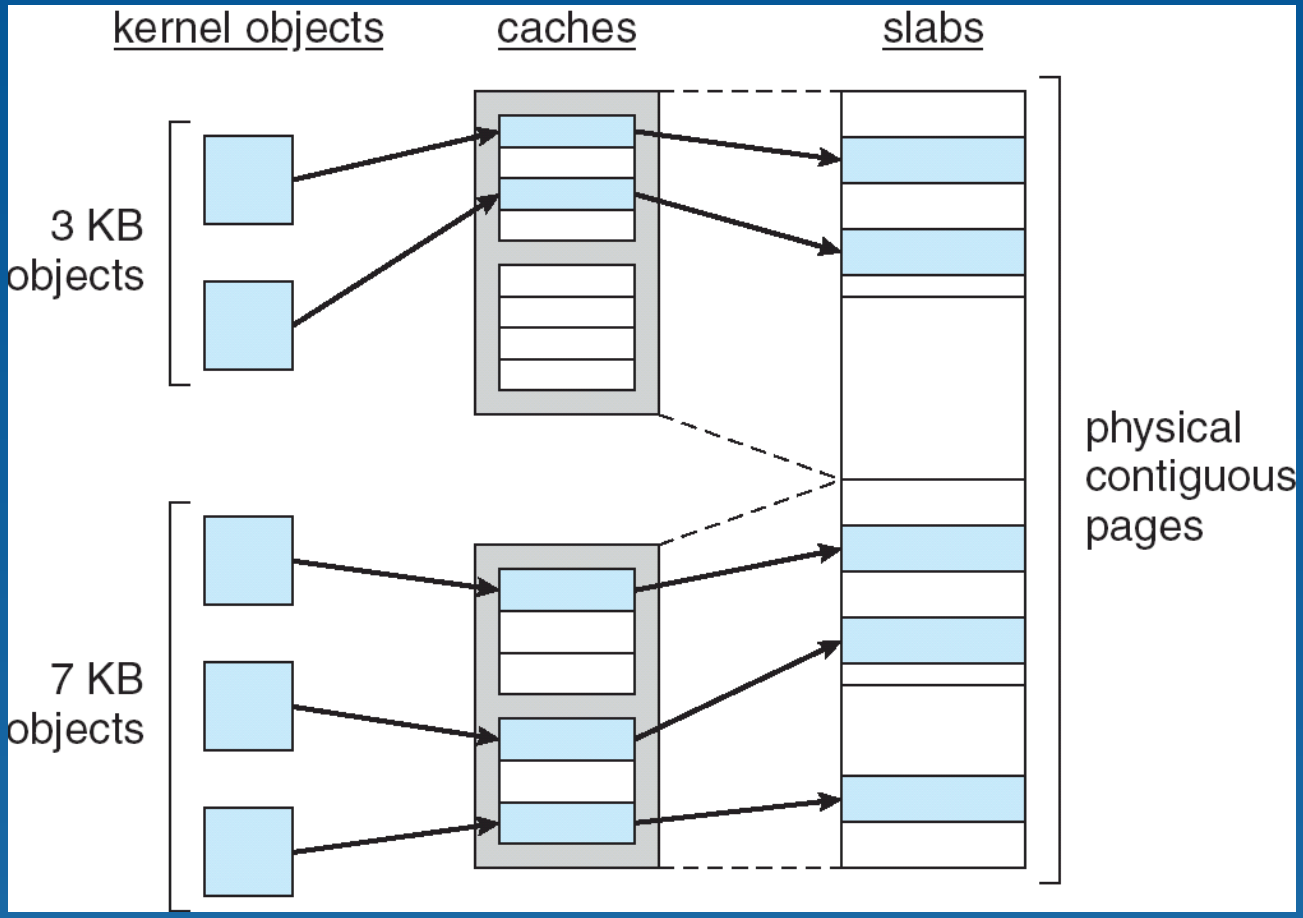
- **Slab** 是一个或数个连续排列的**物理页帧**
- **Cache** 包含了一个或数个 Slabs

- 要求一个 Cache 只包含唯一的一种内核数据结构
  - 一个 Cache 填满了 **objects** – 就是这种内核数据结构的实例

# Slab 算法

- Cache 刚创建时，内含的 objects 标记为 **free**
- 每当内核数据结构被征用了，那个 object 就标记为 **used**
- 如果一个 Cache 用满了，则给 Cache 添加一个空闲 slab 。这个空闲 slab ， 可以容纳一堆 free objects
- Slab 算法优势
- 没有碎片
- 快速响应内核内存请求

# 示例： Slab 算法


kernel objects
caches
slabs
3 KB objects
7 KB objects
physical contiguous pages

# 内核态虚存的申请与释放

- 虚拟空间在3G+high_memory+HOLE_8M以上高端
- 由 vmlist 链表管理
- 用户态内存的申请与释放(mm/vmalloc.c)
  - vmalloc()
  - vfree()

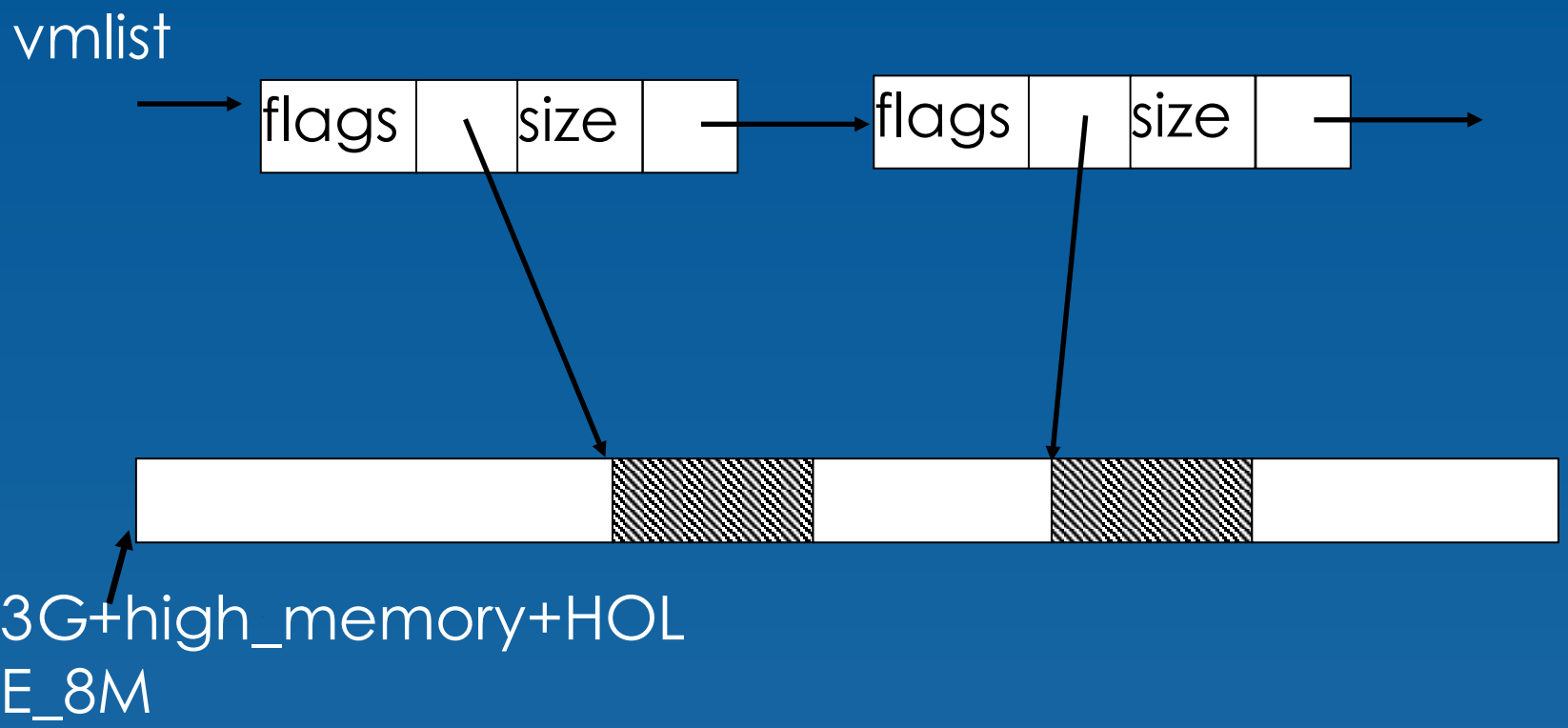
# 内核态虚存的申请与释放

- 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



# 页交换进程和页面换出

- **内核态交换进程 kswapd**
- 一个是内核态线程（ kernel thread ）： 没有虚拟存储空间，运行在内核态，直接使用物理地址空间。
- 它不仅能将页面换出到交换空间（交换区或交换文件），它也保证系统中有足够的空闲页面以保持存储系统高效地运行。
- 在系统初启时由核心态线程 init 创建，并等待系统交换定时器 swap_tick 周期性地唤醒。

# 页交换进程和页面换出

- **系统的空闲页面不够时， kswapd 依次从三条途径缩减系统使用的物理页面**
- 缩减 Page Cache 和 Buffer Cache
- 换出 System V 共享系统的内存页面
- 换出或丢弃进程页面

# 缺页中断和页面换入

- 产生缺页的虚存地址（CR2）传递给内核的缺页中断服务程序
- 如果没有找到与缺页相对应的vm_area_struct 结构，那么说明进程访问了一个非法存储区， Linux 内核向进程发送信号SIGSEGV
- 接着检查访问权限，看缺页该访问是否合法
- **经过以上两步检查，可以确定的确是缺页中断**

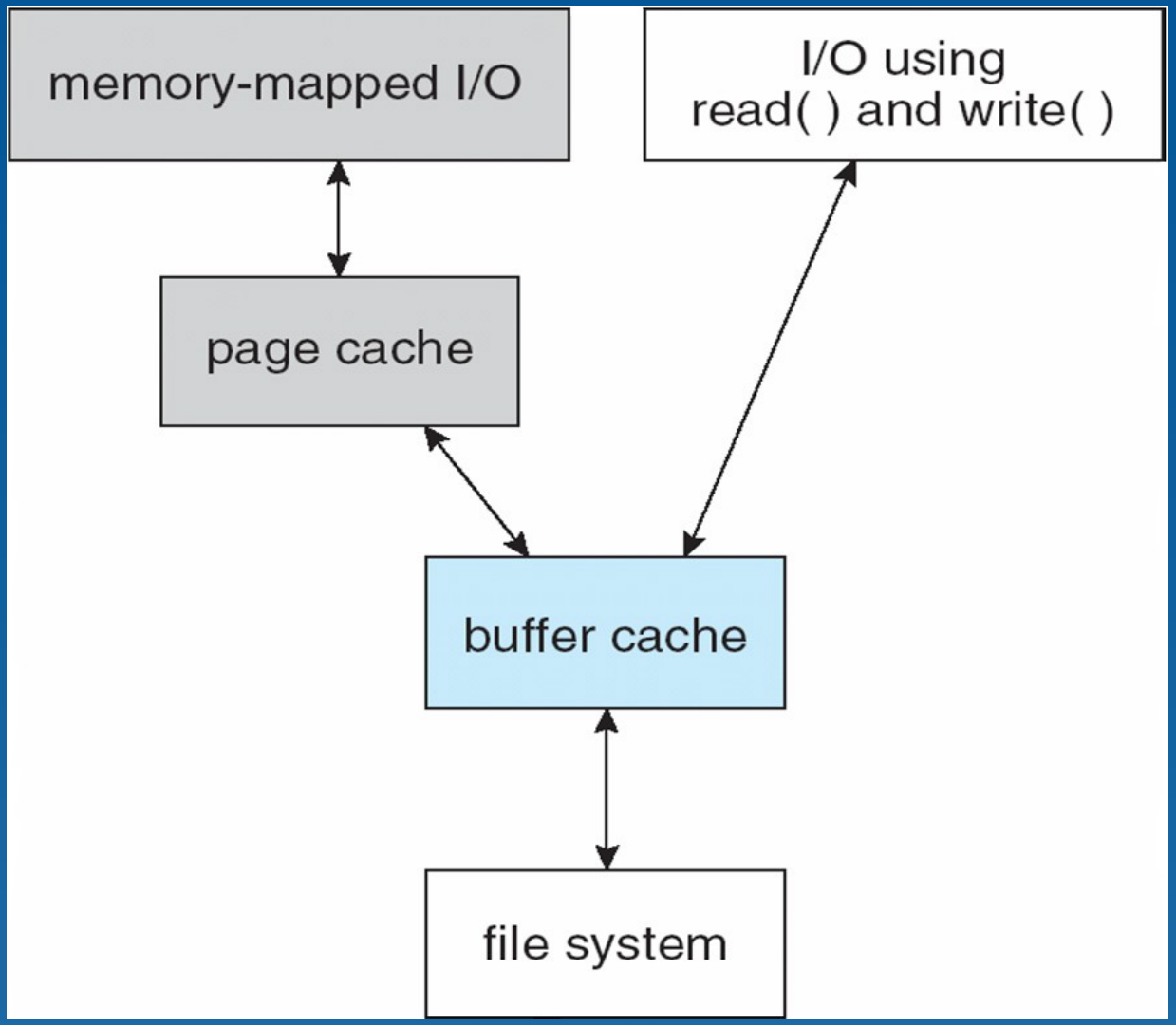
# Cache

- Swap Cache
  - Swap Cache 是一个页表项的列表，每一项与内存中的一帧相对应
  - 被换出的页面在 Swap Cache 中占有一表项，该表项描述页面在交换空间中的位置
  - 当该页面修改时，该表项将被清除
- Page Cache
- Buffer Cache

# Page Cache & Buffer Cache

- page cache 在虚拟内存空间直接 cache 页面，而不是 cache 磁盘数据块
- Memory-mapped I/O 技术使用 page cache
- 文件系统的 I/O 例程使用 buffer cache

# I/O 操作使用两种 Cache




End