Lab2 introduction

2010-04-30

Lab2任务清单

- 实现物理页面管理
- 掌握段页式地址映射机制
- 掌握内核虚拟内存布局
- 建立二级页表
- 掌握bochs针对不同类地址的调试命令

Lab2准备

- 将Lab1的解答放入Lab2
  - 取得Lab1修改内容
    - diff -u -r Lab1 Lab1-change > Lab1-changes.patch
  - 将Lab1的修改patch到Lab2
    - cd Lab2
    - patch -p1 -u < ../Lab1-changes.patch
  - 将*.rej中未加入的更改手动加入相应文件

outline(1)

- Lab2代码树
- Introduction
  - 符号表结构
- Part 1 Physical Page Management
  - struct Page数据结构
  - queue.h中实现的双向链表
  - 过渡阶段的页目录

outline(2)

- **Part 2 Virtual Memory**
  - 段页式映射机制
  - 逻辑地址、线性地址、物理地址
- 重要数据结构和函数
- **Part 3: Kernel Address Space**
  - 段页映射中的权限检查
  - 自映射
  - 过渡阶段的页目录

Lab2 代码树

lab2

- **boot**: 引导扇区代码
- **CODING**: 代码规范说明;
- **Conf, GNUmakefile, mergedep.pl**: 编译相关文件;
- **grade.sh**: 代码测试脚本;
- **inc**: 头文件定义***
- **kern**: 内核代码***
- **lib**: 代码库;
- **user**: 用户态程序; 了解

Lab2 代码树： inc

inc

memlayout.h***： 内存管理相关的宏定义；其中注释中的虚拟内存表是重中之重

mmu.h**:MMU相关的宏定义；

Lab2 代码树： kern

- kern
  - pmap.h***: 一定要在读pmap.c前通读一遍
  - pmap.c***: 内存管理
  - kclock.h
  - kclock.c： PC时钟控制
  - kdebug.h*
  - kdebug.c**: debug信息

outline

- **Introduction**
  - 符号表结构

Introduction—Ex1

- **Exercise1：**
  - 扩展Lab1中的Stack Backtrace功能，通过调用stab_binsearch 与read_eip()实现debuginfo_eip()函数，使原先显示eip的位置变为所在的函数名称和偏移量

```
Stack backtrace:
kern/monitor.c:74: mon_backtrace+10
  ebp f0119ef8  eip f01008ce  args 00000001 f0119f20 00000000 00000000 2000000a
kern/monitor.c:143: monitor+10a
  ebp f0119f78  eip f01000e5  args 00000000 f0119fac 00000275 f01033cc ffffffc
kern/init.c:78: _panic+51
  ebp f0119f98  eip f010133e  args f01033ab 00000275 f01033cc f0103473 f01030bc
kern/pmap.c:711: page_check+9e
  ebp f0119fd8  eip f0100082  args f0102d20 00001aac 000006a0 00000000 00000000
kern/init.c:36: i386_init+42
  ebp f0119ff8  eip f010003d  args 00000000 00000000 0000ffff 10cf9a00 0000ffff
```

实现原理

- JOS内核中有若干段
- .stab段是内核符号表，其中存有文件、函数、行数等的信息； .stabstr段中存有相应的文件和函数名等

```
Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .text         00003c00  f0100000  f0100000  00001000  2**4
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  1 .rodata       00000ef8  f0103c00  f0103c00  00004c00  2**5
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  2 .stab         00005779  f0104af8  f0104af8  00005af8  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  3 .stabstr      00002656  f010a271  f010a271  0000b271  2**0
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  4 .data         00008358  f010d000  f010d000  0000e000  2**12
                  CONTENTS, ALLOC, LOAD, DATA
  5 .bss          00000690  f0115360  f0115360  00016358  2**5
                  ALLOC
  6 .comment      00000222  00000000  00000000  00016358  2**0
                  CONTENTS, READONLY
```

实现原理

- backtrace需要从.stab段和.stabstr段中获取所需的信息：
  - 文件名
  - 函数名
  - 行号
  - 调用点在函数中的偏移量等
- 由debuginfo_eip()实现这些功能
- backtrace通过调用debuginfo_eip()来获得所需信息

实现原理：Stab结构

- **struct Stab**
  - **n_strx**：指向stabstr表的索引，stabstr表中存有该符号项的名字
  - **n_type**：该符号项的类型
  - **n_other**：杂项信息（通常为空）
  - **n_desc**：描述信息
  - **n_value**：该符号项的值

```
Symnum n_type n_othr n_desc n_value  n_strx String
1624   SO     0      2      f0103570 9116   lib/string.c
1646   FUN    0      0      f0103570 9129   strlen:F(0,1)
1647   PSYM   0      0      00000008 9143   s:p(0,19)=*(0,2)
1648   SLINE  0      7      00000000 0
1649   SLINE  0      10     00000006 0
1650   SLINE  0      11     00000015 0
1651   SLINE  0      10     00000018 0
1652   SLINE  0      13     0000001e 0
1653   RSYM   0      0      00000000 8849   n:r(0,1)
```

实现原理

- n_type说明：
  - N_SO：表示文件
  - N_SOL：被包含的文件
  - N_FUN：表示函数
  - N_SLINE:表示在文件中的行号
- 文件和函数项的n_value字段表示该文件或函数装载后的虚拟地址
- n_strx是指向.stabstr段的索引

实现过程

- 通过objdump命令，观察内核中不同的段。
- objdump -h obj/kern/kernel
  - 需要注意.stab和.stabstr两段
- objdump -G obj/kern/kernel > stabs.txt
  - 由于显示内容较多，可以将结果输出到文件中
  - 文件(N_SO)和函数
    址递增的顺序组织

```
For example, given these N_SO stabs:
    Index  Type   Address
    0      SO     f0100000
    13     SO     f0100040
    117    SO     f0100176
    118    SO     f0100178
    555    SO     f0100652
    556    SO     f0100654
    657    SO     f0100849
```

实现过程

- 根据eip和n_type(N_SO,N_SOL或N_FUN)，在.stab段中查找相应的Stab项（通过调用stab_binsearch）
- 根据相应Stab项的n_strx域，找到该项在.stabstr段中的索引，从该索引开始的字符串就是该项的名字（文件名或函数名）
- 根据eip和n_type(N_SLINE)，在.stab段中找到相应的行号(n_desc字段)

backtrace阅读资料

- 代码阅读：
  - inc/elf.h, inc/stab.h, kern/kdebug.c
- 参考资料：
  - elf文件格式：http://os.pku.edu.cn:8080/gaikuang/files/09ospro/readings/elf.pdf
  - stabs文档：http://sources.redhat.com/gdb/onlinedocs/stabs_toc.html（local copy on os.pku.edu.cn）

outline

- **Part 1 Physical Page Management**
  - **struct Page数据结构**
  - **queue.h中实现的双向链表**

Part2--Ex2

- **Exercise2：**
  - 完成函数boot_alloc()、page_init()、page_alloc()、page_free()，实现对物理内存页面的管理
- 知识点
  - struct Page
  - queue.h中的链表

Struct Page数据结构

- Page结构说明
  - Page_LIST_entry_t pp_link
    - 用于page链表管理
  - uint16_t pp_ref
    - 该物理页面被引用数（即被map到虚拟地址的数量）
    - 当引用数为0，即可释放

inc/queue.h

- 提供了进行“双向”链表定义和操作的宏
- 包括两组宏
  - 一组定义类型
    - **LIST_ENTRY**
    - **LIST_HEAD**
  - 一组定义操作
    - **LIST_INIT**
    - **LIST_FOREACH**
    - **LIST_INSERT_HEAD**
    - **LIST_REMOVE**

LIST_ENTRY

```
#define LIST_ENTRY(type)                                        \
struct {                                                        \
        struct type *le_next;   /* next element */              \
        struct type **le_prev;  /* ptr to ptr to this element */    \
}
```

- 定义结点的指针部分
  - **Type是链表结点的类型**
  - **LIST_ENTRY(type)定义type类型包含的两个指针**
  - 这两个指针把结点串接成双向链表

一个需要注意的地方

- le_next指向下一个结点
- le_prev指向上一个结点里的le_next属性
  - 这样做与让le_prev直接指向上一结点等效
  - 但是能让表头元素处理起来更简单

在lab2中的使用

```
typedef LIST_ENTRY(Page) Page_LIST_entry_t;  #inc/memlayout.h

struct Page {
        Page_LIST_entry_t pp_link;      /* free list link */
        uint16_t pp_ref;
};
```

LIST_HEAD

```
#define LIST_HEAD(name, type)                                   \
struct name {                                                   \
        struct type *lh_first;  /* first element */             \
}
```

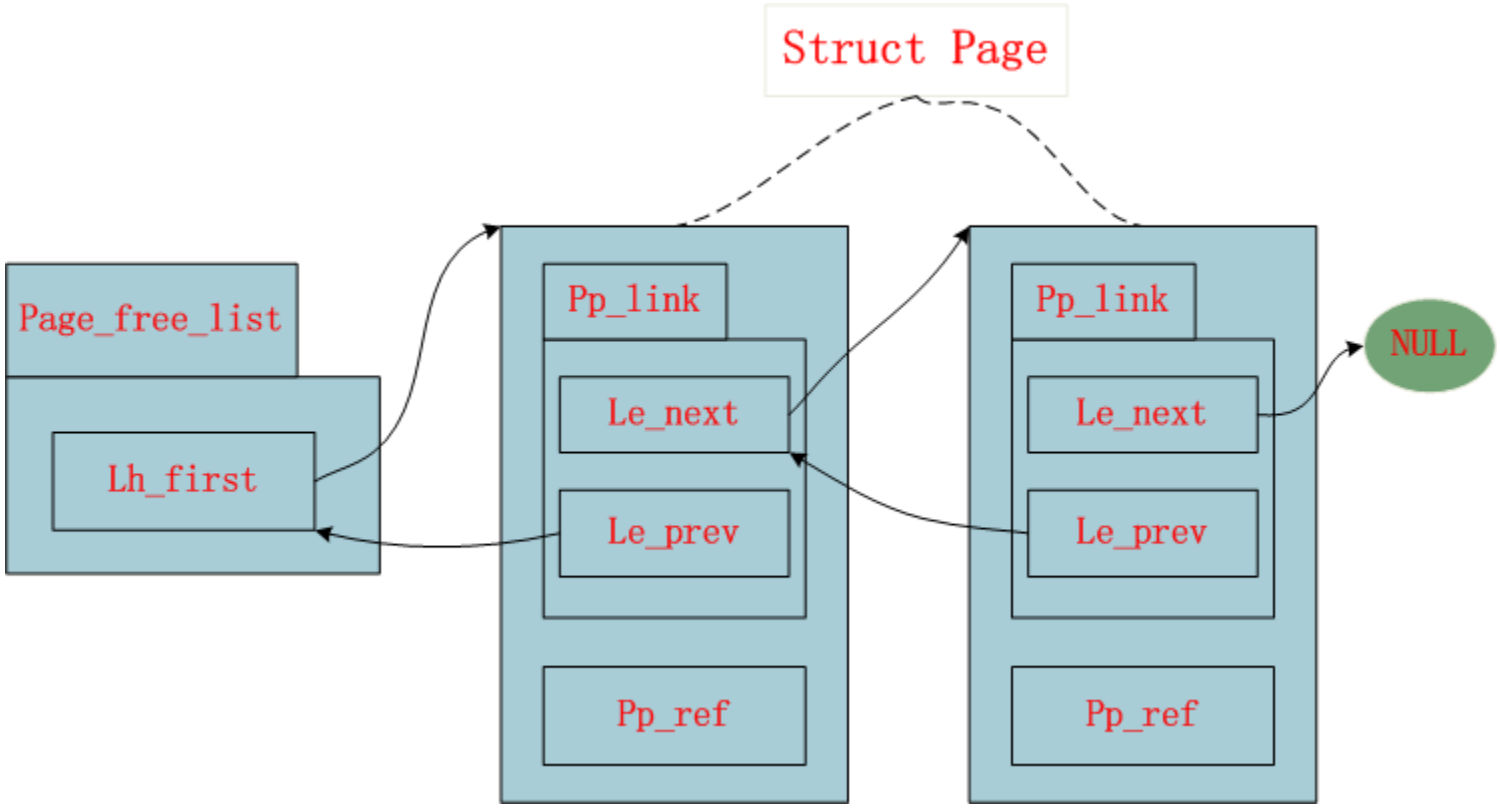
- 这个宏实现了类型名是**name**，结点类型是**type**的双向链表的结构体定义。

在lab2中的使用

LIST_HEAD(Page_list, Page); #inc/memlayout.h
static struct Page_list page_free_list; #kern/pmap.c

page_free_list被用来管理空闲的物理页面。

Page_free_list示例



定义操作的宏

- 链表初始化
  - #define LIST_INIT ...
- 遍历链表中的所有结点
  - #define LIST_FOREACH ...
- 在链表的表头位置添加一个结点
  - #define LIST_INSERT_HEAD ...
- 从链表中删除一个结点
  - #define LIST_REMOVE ...

outline

- **Part 2 Virtual Memory**
  - 段页式映射机制
  - 逻辑地址、线性地址、物理地址

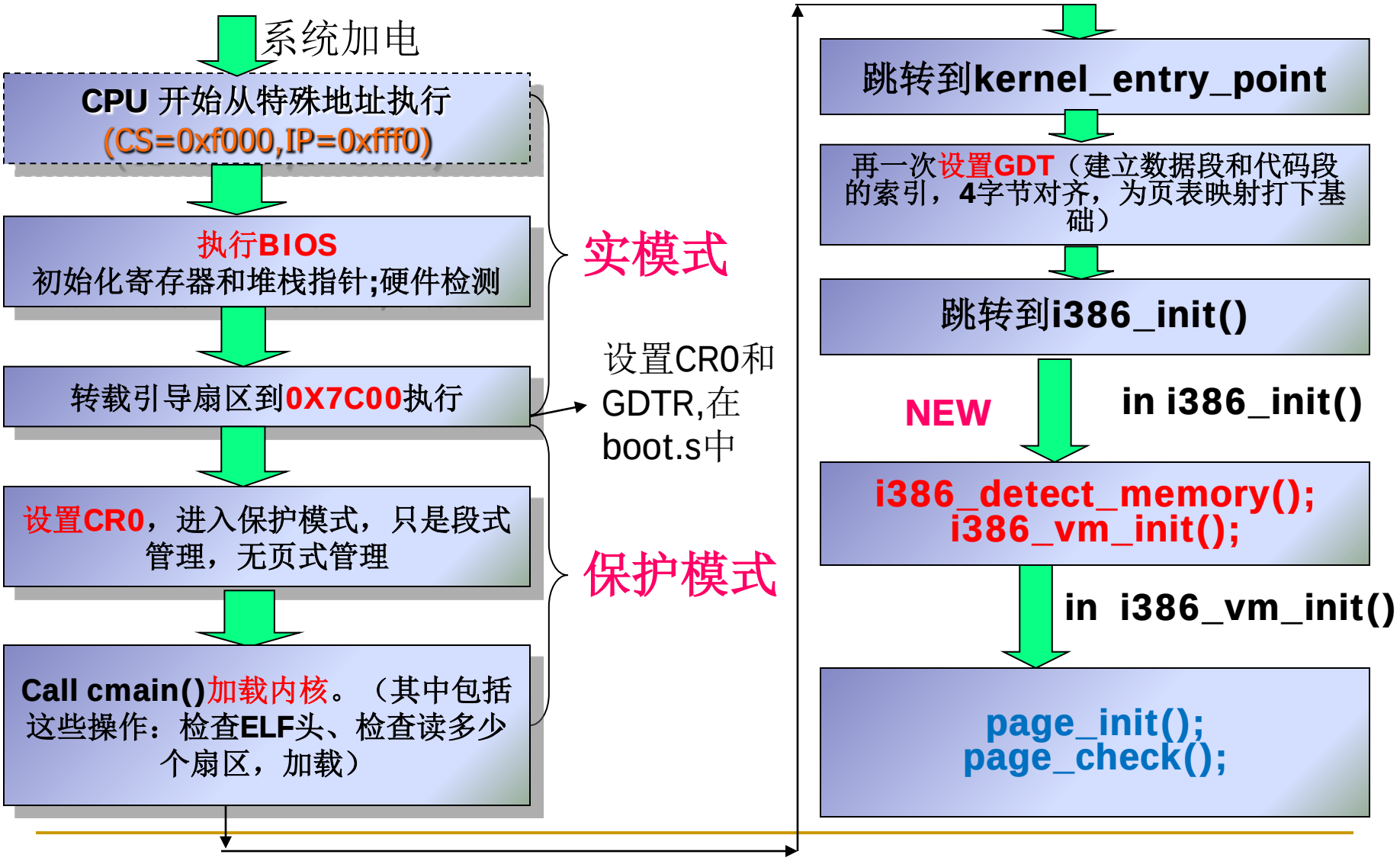
Part2—Ex3、4

- **Exercise3：**
  - 阅读Intel手册，了解段式映射在保护模式下的使用
- **Exercise4：**
  - 阅读Bochs手册中关于内存内容显示的功能介绍，区分逻辑/线性/物理地址，熟练使用命令来验证自己的程序

知识点

- 开机启动时内存管理流程
- 虚拟内存布局
- 逻辑地址 vs. 线性地址 vs. 物理地址
- 保护模式下地址转换流程
- 段页式地址转换流程
- 页目录项和页表项结构
- TLB

启动时内存管理流程


系统加电
CPU 开始从特殊地址执行
(CS=0xf000,IP=0xfff0)
执行BIOS
初始化寄存器和堆栈指针;硬件检测
转载引导扇区到0X7C00执行
设置CR0，进入保护模式，只是段式管理，无页式管理
Call cmain()加载内核。（其中包括这些操作：检查ELF头、检查读多少个扇区，加载）
实模式
保护模式
设置CR0和GDTR,在boot.s中
跳转到kernel_entry_point
再一次设置GDT（建立数据段和代码段的索引，4字节对齐，为页表映射打下基础）
跳转到i386_init()
NEW
in i386_init()
i386_detect_memory();
i386_vm_init();
in i386_vm_init()
page_init();
page_check();

虚拟内存布局及相关的宏(1)

```
/* Virtual memory map:                                Permissions
 *                                                    kernel/user
 *        4 Gig --> +----------------------------+
 *                  |  Remapped Physical Memory  | RW/--
 *     KERNBASE --> +----------------------------+ 0xf0000000
 *                  | Cur. Page Table (Kern. RW) | RW/--  PTSIZE
 *VPT,KSTACKTOP --> +----------------------------+ 0xefc00000    --+
 *                  |        Kernel Stack        | RW/--  KSTKSIZE |
 *                  | - - - - - - - - - - - - - -|                 PTSIZE
 *                  |     Invalid Memory (*)     | --/--           |
 *         ULIM --> +----------------------------+ 0xef800000    --+
 *                  | Cur. Page Table (User R-)  | R-/R-  PTSIZE
 *         UVPT --> +----------------------------+ 0xef400000
 *                  |         RO PAGES           | R-/R-  PTSIZE
 *       UPAGES --> +----------------------------+ 0xef000000
 *                  |                            |
 *                  |      Empty Memory (*)      |
 *                  |                            |
 *            0 --> +----------------------------+
 */
```

内存布局及相关的宏(2)

- **KERNBASE**
  - 内核逻辑地址的起始点。从KERNBASE到4G的逻辑地址映射了0-256M的物理内存，以方便内核直接访问
- **ULIM**
  - 用户态程序可以访问地址的界限，更高的内存用户不可读。一般用来方便判断用户访存是否超界
- **UTOP**
  - 用户有写权限的地址界限。UTOP和ULIM之间是用户只读的内核数据结构，如UVPT，UPAGES

内存布局及相关的宏(3)

- **VPT/UVPT**
  - 当前用户进程/内核的页表项目映射的位置，其中每个4K页面对应一个页目录项中的一个页表
  - UVPT是为用户程序只读访问VPT而做的映射
- **UPAGES**
  - Page结构数组在内存中的映射，其中每个page结构记录了一个物理页面是否被使用以及使用次数、在空闲页链表中使用的链接等信息

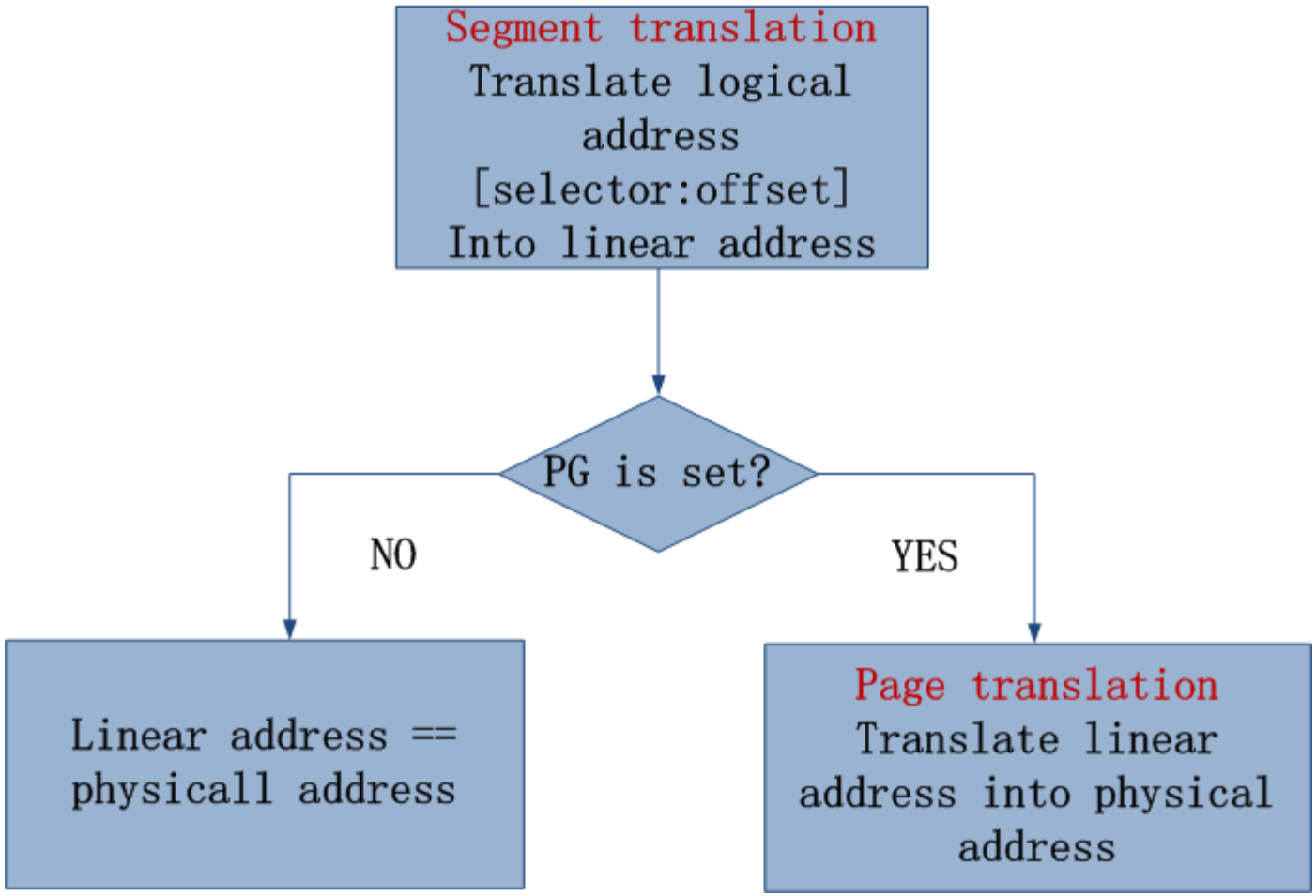
逻辑地址、线性地址、物理地址（1）

- 逻辑地址
  - 指令中用来指定一个操作数或一条指令的地址
  - 由段选择符和偏移量组成
    - **Segment selector:offset**
- 线性地址
  - 逻辑地址经过段式机制转换后，尚未进行页式转换的地址
- 物理地址
  - 逻辑地址经过段式和页式机制转换后得到的地址
  - 用于内存芯片级内存单元寻址，和电信号对应

逻辑地址、线性地址、物理地址（2）

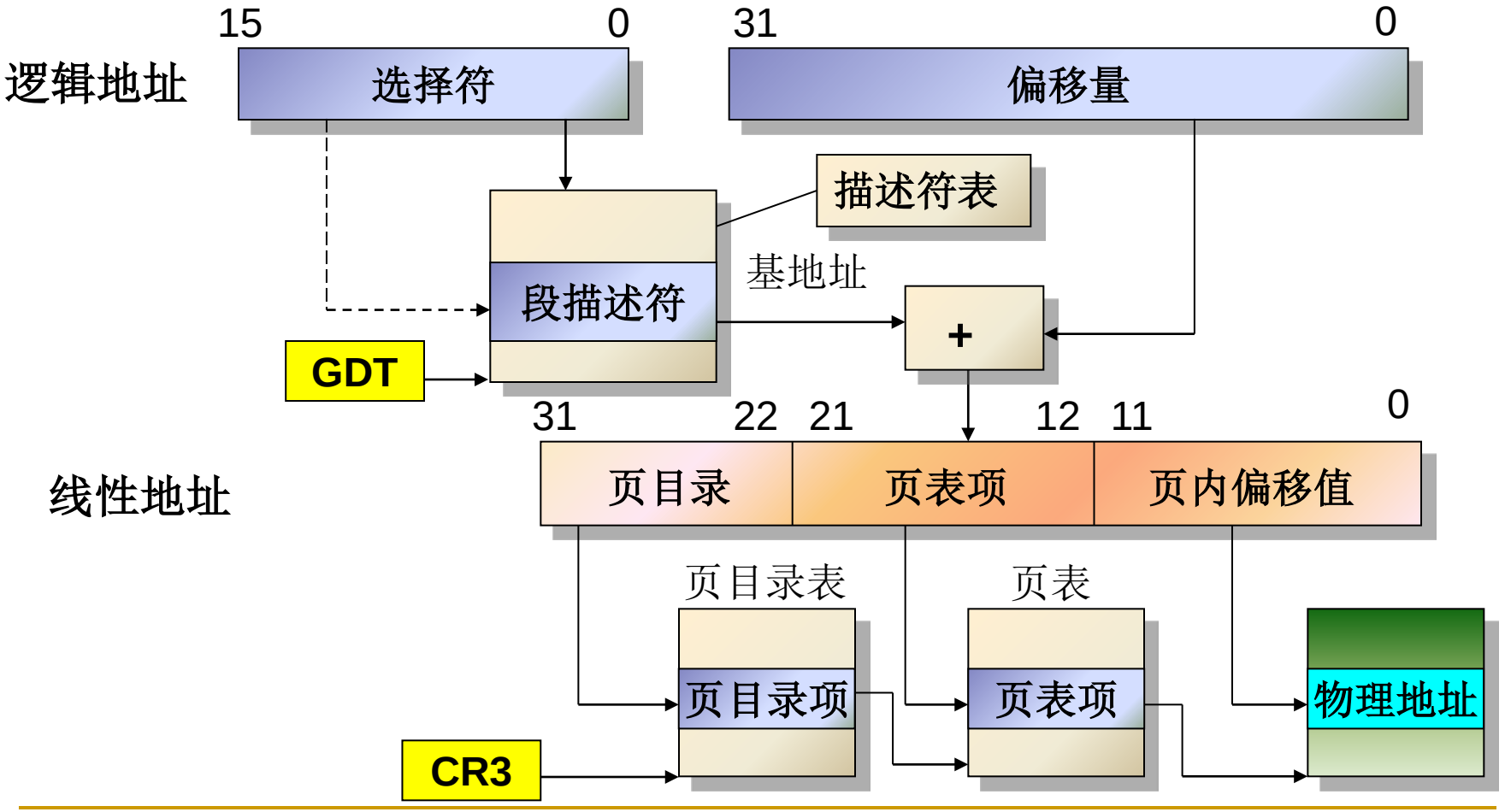
- Bochs设置断点
  - vb：在逻辑地址处设置
  - lb：在线性地址处设置
  - b/pb：在物理地址处设置
- 查看内存内容
  - x：线性地址
  - xp：物理地址

保护模式下地址转换流程



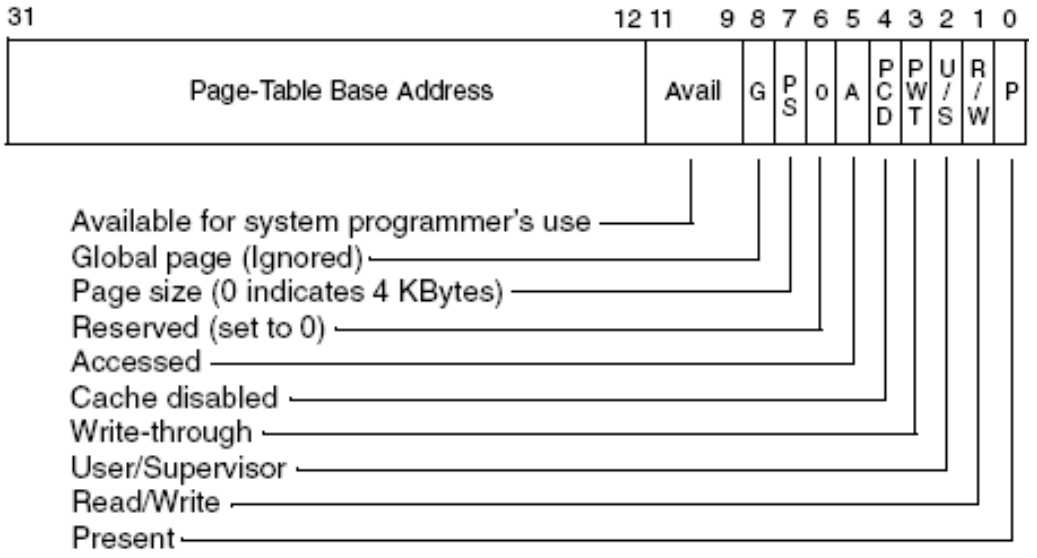
PG是控制寄存器cr0的一位

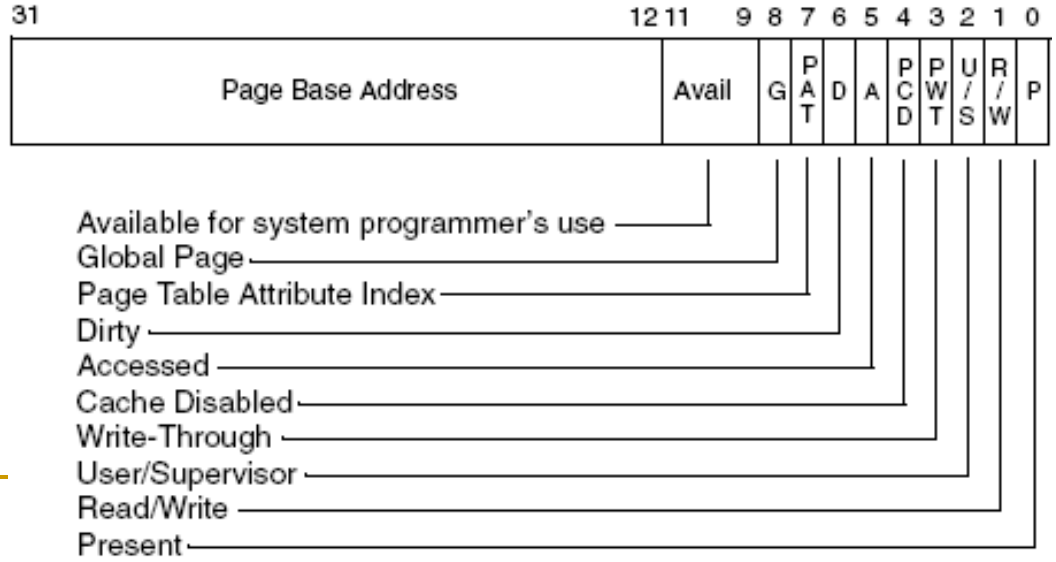
段页式内存映射机制


逻辑地址
15
0
选择符
31
0
偏移量
描述符表
段描述符
基地址
+
GDT
线性地址
31
22
21
12
11
0
页目录
页表项
页内偏移值
页目录表
页表
页目录项
页表项
物理地址
CR3

页映射中的数据与意义

- 页目录项
  - **4M页Challenge**
  - 调试时注意检查此处的权限
- 页表项





TLB

- TLB (translation lookaside buffer)
  - 缓存了线性地址到物理地址的映射的一部分
  - 访问TLB比访问内存快
  - 在TLB和页目录/页表中并行查找提高页式地址转换的效率
  - 当页目录项或页表项被修改或删除时，对应的TLB项(如果存在)必须被删除来保持一致(调用tlb_invalidate)
  - 写cr3寄存器会导致TLB被清空

outline

- 重要数据结构和函数

重要函数、数据结构(1)

- 在内核虚拟内存布局中，需要设置内核数据结构以及堆栈的映射，注意以下宏的定义和使用
  - boostack
  - Kernbase
  - VPT
  - UENVS(user readable)
  - UPAGES(user readable)
- 注意inc/memlayout.h中的内存布局

重要函数、数据结构(2)

- 二级页表映射的建立
  - 阅读mmu.h文件，熟悉一些PDX和PTX的宏的定义
  - 阅读kern/pmap.h文件，了解如下函数
    - Pa2page—从物理地址得到page结构
    - Page2pa—从page结构得到物理地址
    - Page2ppn—从page结构得到得到物理页号
    - PADDR—虚拟地址转换成物理地址
    - KADDR—物理地址转换成虚拟地址
    - Page2kva—从page结构得到得到虚拟地址

重要函数、数据结构(3)

- Page_init()初始化函数，该函数的编写重点是要找到内核区域

  注：该函数以后还需要修改

- 如果熟悉freelist的一些链式操作，那么编写这些函数不是一个很困难的事情

  注：在编写时，需要把页面访问的权限放大，这样会减少将来Lab调试的很多麻烦。必要时设置用户态可读

重要函数、数据结构(4)

- 注意kern/pmap.c中
  - pgdir[0] = pgdir[PDX(KERNBASE)];
  - pgdir[0] = 0;

  作用分析
  - 内核的Link Address是如何与Load Address匹配的
  - 在此之前两个地址是否相等？
  - 如果没有，代码中做了哪些工作来解决？
  - 在内核两个地址不相等时，做什么操作会发生问题？

outline

- **Part 3: Kernel Address Space**
  - 段页映射中的权限检查
  - 自映射
  - 过渡阶段的页目录

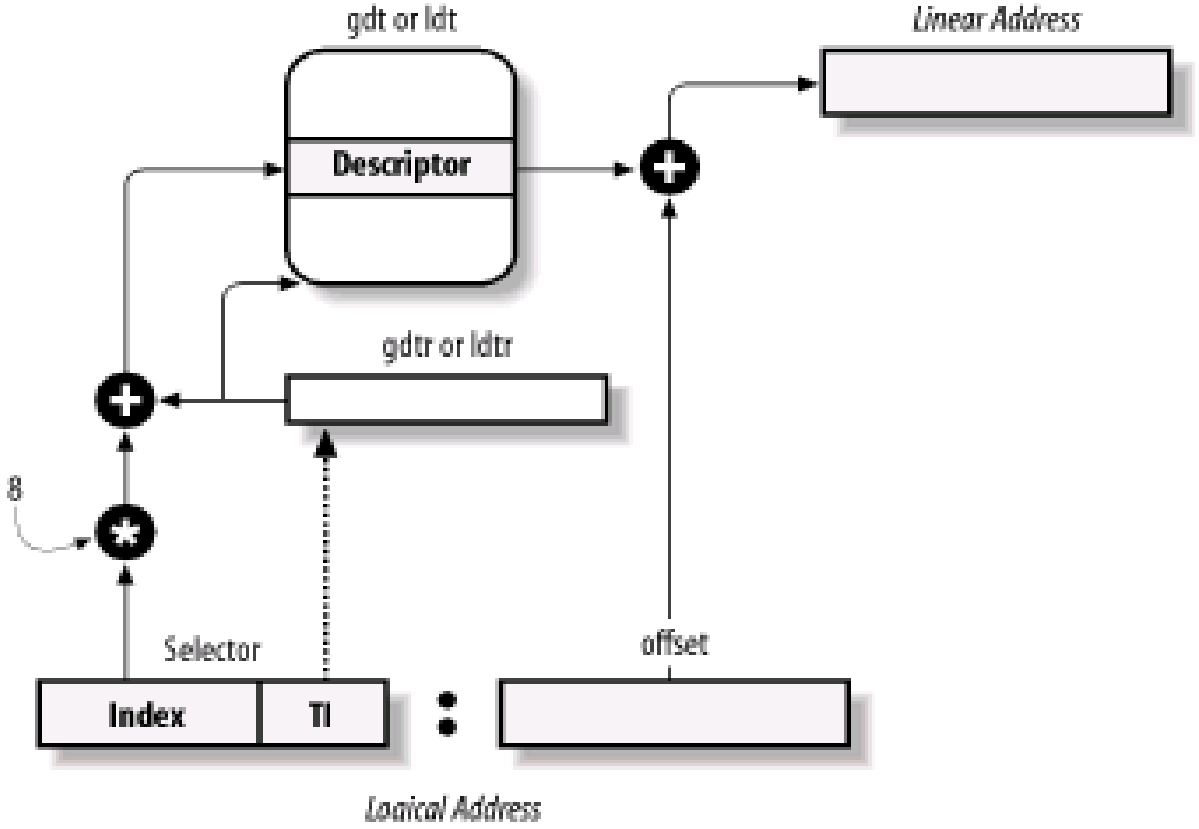
权限检查-基本概念

基本概念

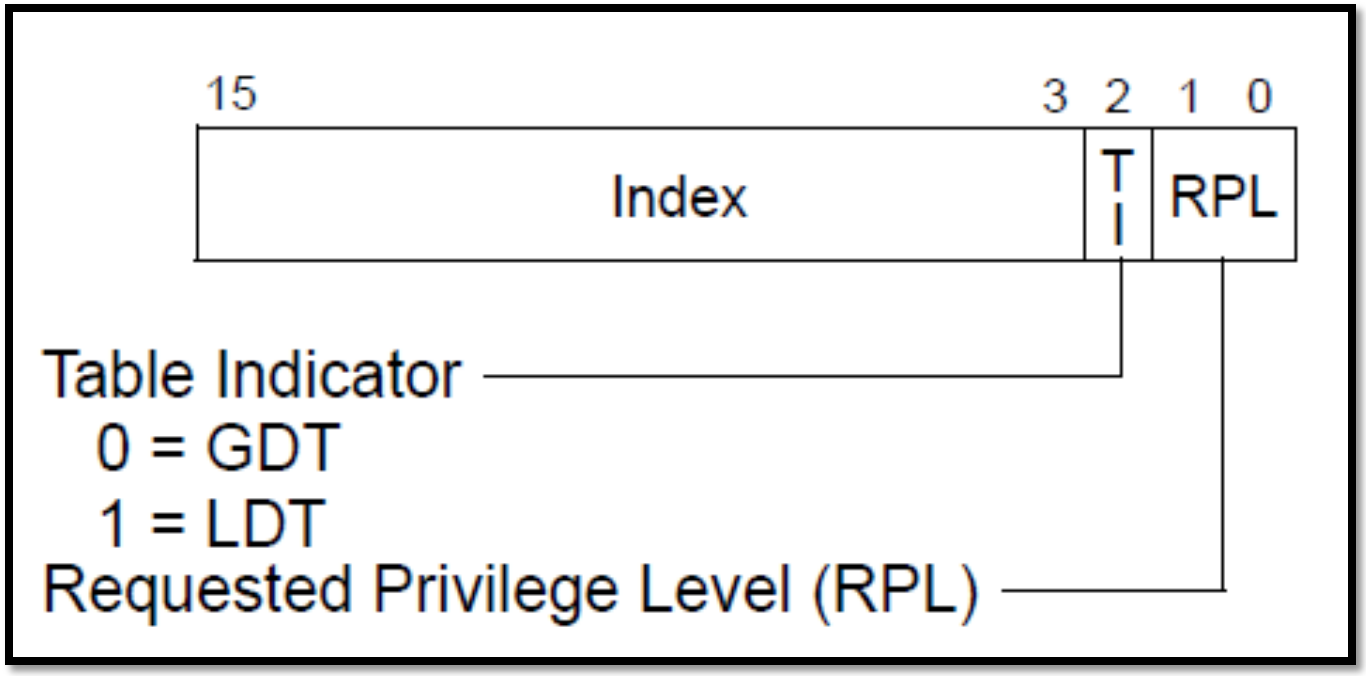
- **Current privilege level (CPL) — 当前运行进程的权限级别**
- **Descriptor privilege level (DPL) — 要访问的段的权限级别**
- **Requested privilege level (RPL) — 段选择子（符）的权限级别**

回顾：段映射过程

- 逻辑地址生成线性地址
- 区分全局段描述符gdtr和本地段描述符ldtr



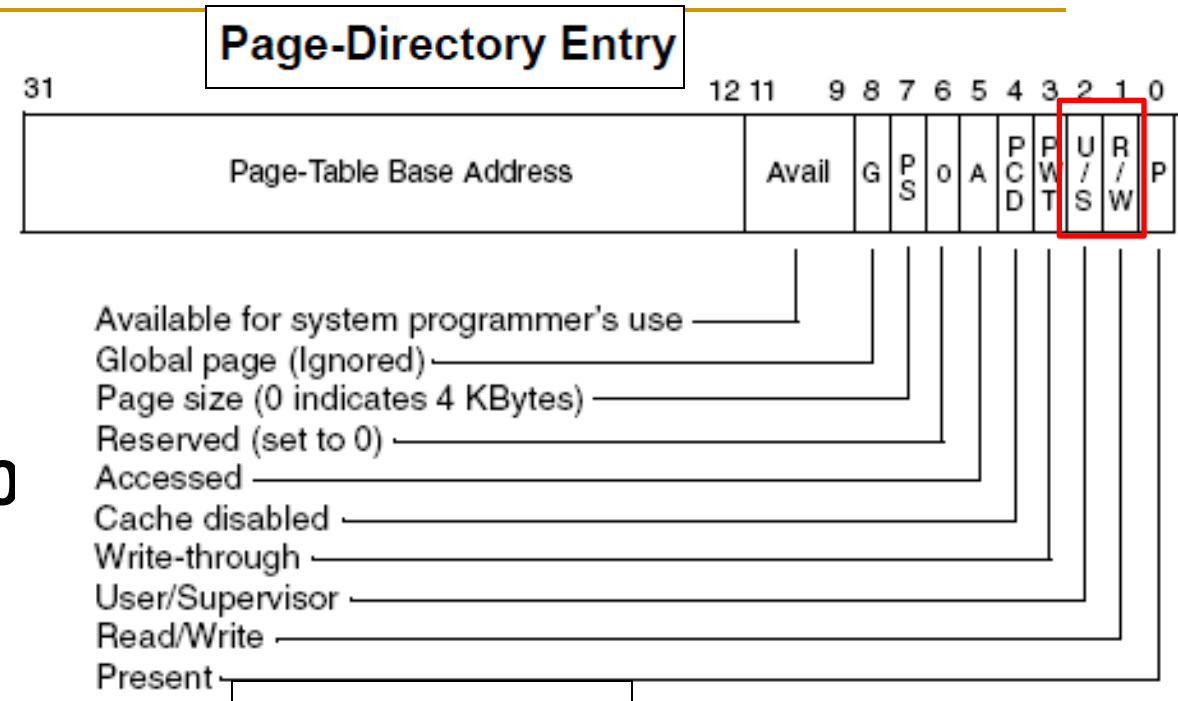
段映射-权限位

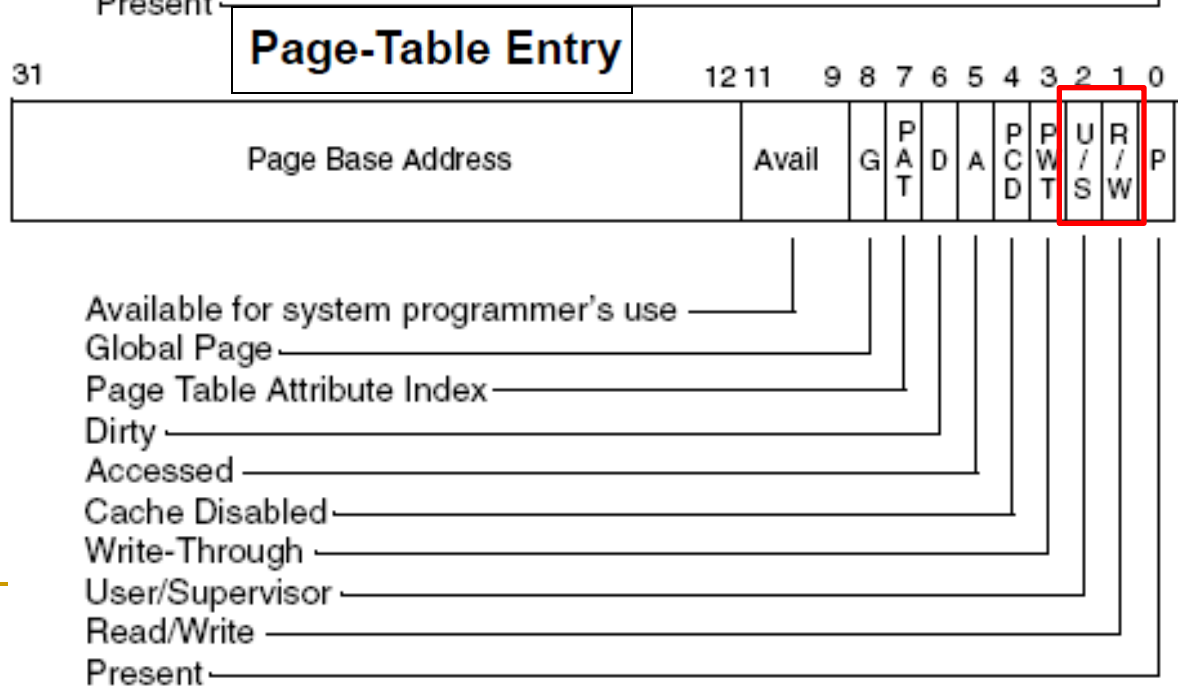


Segment Selector

页映射-权限位

- U/S为1表示用户可访问(CPL1,2,3)，为0表示只有内核可访问(CPL0)
- R/W表示读写权限，1表示可写，0表示可读





段映射权限检查步骤

- 以访问数据段为例
  - ★ CPL（当前程序运行的权限级别）与RPL（位于selector中的 RPL）作比较，得出的有效权限级别为低权限的一个
  - ★ 得出的有效权限级别与DPL（segment descriptor 中的DPL）作比较，有效权限级别高于DPL，那么就通过。低于就不允许访问
- 例：在访问data segement时*

  CPL=1,RPL=2,DPL=3: 能否访问？

段映射权限检查步骤*

- **DPL(Descriptor Privilege Level)**

  表示门或者段的特权级根据段或者门的类型不同，DPL的含义不同：

  - 1.数据段的DPL：规定了访问此段的最低权限。比如一个数据段的DPL是1，那么只有MAX（ CPL ,RPL）为0或1的程序才可能访问它
  - 2.非一致代码段的DPL(不使用调用门的情况)？
  - 3.调用门的DPL？
  - 4.一致代码段和通过调用门访问的非一致代码段？
  - 5. TSS的DPL？

段映射权限检查步骤*

- **DPL(Descriptor Privilege Level)**

  表示门或者段的特权级根据段或者门的类型不同，DPL的含义不同：

  - 1.数据段的DPL：规定了访问此段的最低权限。比如一个数据段的DPL是1，那么只有MAX（ CPL ,RPL）为0或1的程序才可能访问它
  - 2.非一致代码段的DPL(不使用调用门的情况)？
  - 3.调用门的DPL？
  - 4.一致代码段和通过调用门访问的非一致代码段？
  - 5. TSS的DPL？

- http://os.pku.edu.cn:8080/gaikuang/files/09ospro/readings/ia32/IA32-3A.pdf

页映射权限检查步骤

LAB2相关！

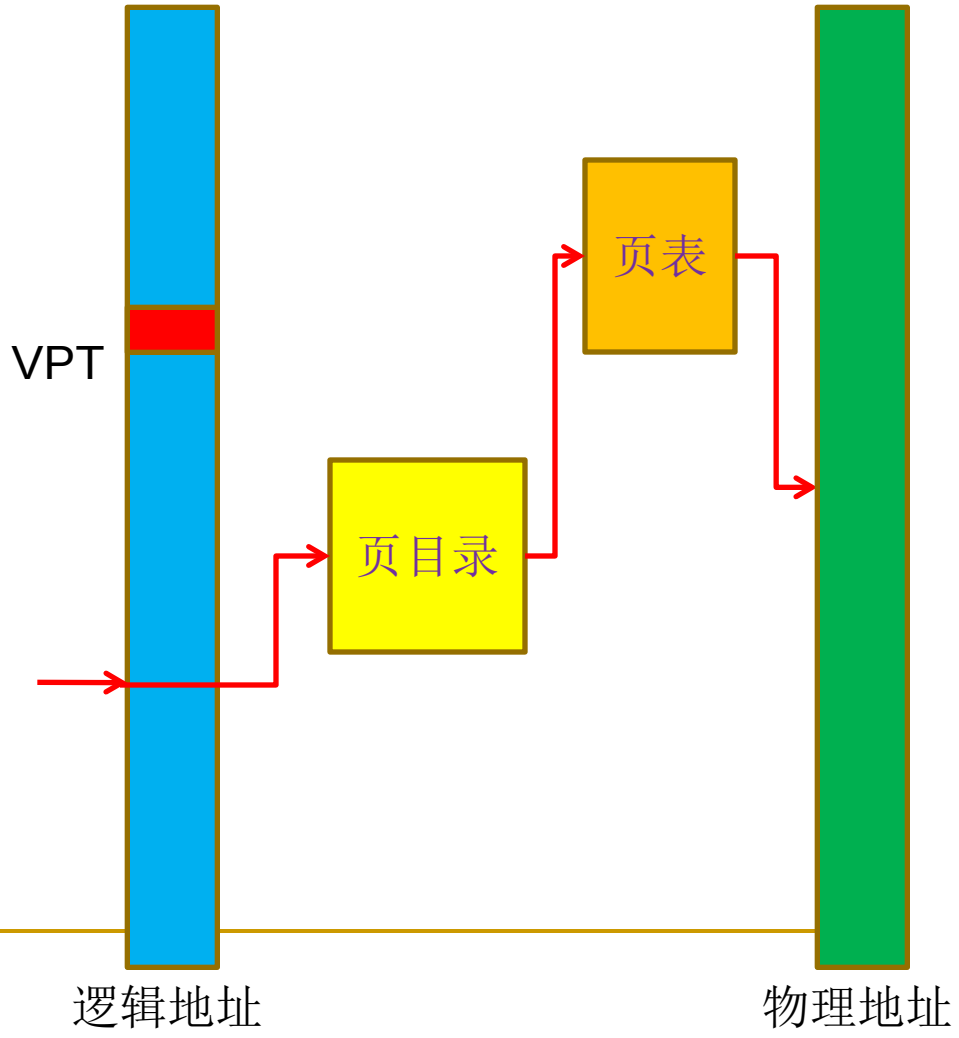
- 得到段映射生成的线性地址
- 确定是否有访问页面的权限
  - 确定是否足以访问页对应的页目录项
    - 判断页目录项PTE_P，是否页面不存在，无法访问
    - 判断页目录项PTE_U，是否只有CPL0可以访问
    - 判断页目录项PTE_W，是否禁止写操作
  - 页表项检查PTE_P, PTE_U, PTE_W方式和页目录项相同
- 根据物理地址访问内存

Virtual Page Table 映射

- 在kern/pmap.c
  - Pgdir[PDX(VPT)] = PADDR(pgdir) | ……
  - 为pgdir建立了VPT位置指向自身的映射
- 无VPT映射时访问修改页表项方式
  - Pgdir[PDX(va)]得到页表的物理地址，转换为逻辑地址，再根据逻辑地址取PTX(va)的偏移量，得到pte
- 映射VPT后访问方式
  - 在逻辑地址VPT开始的4M空间中，即是所有4G空间的pte的排布，根据VPT[PDX(va)*4K+PTX(VA)]即可直接访问一个pte
- 将需要代码实现的逻辑用CPU的mmu实现，提高效率，方便代码编写

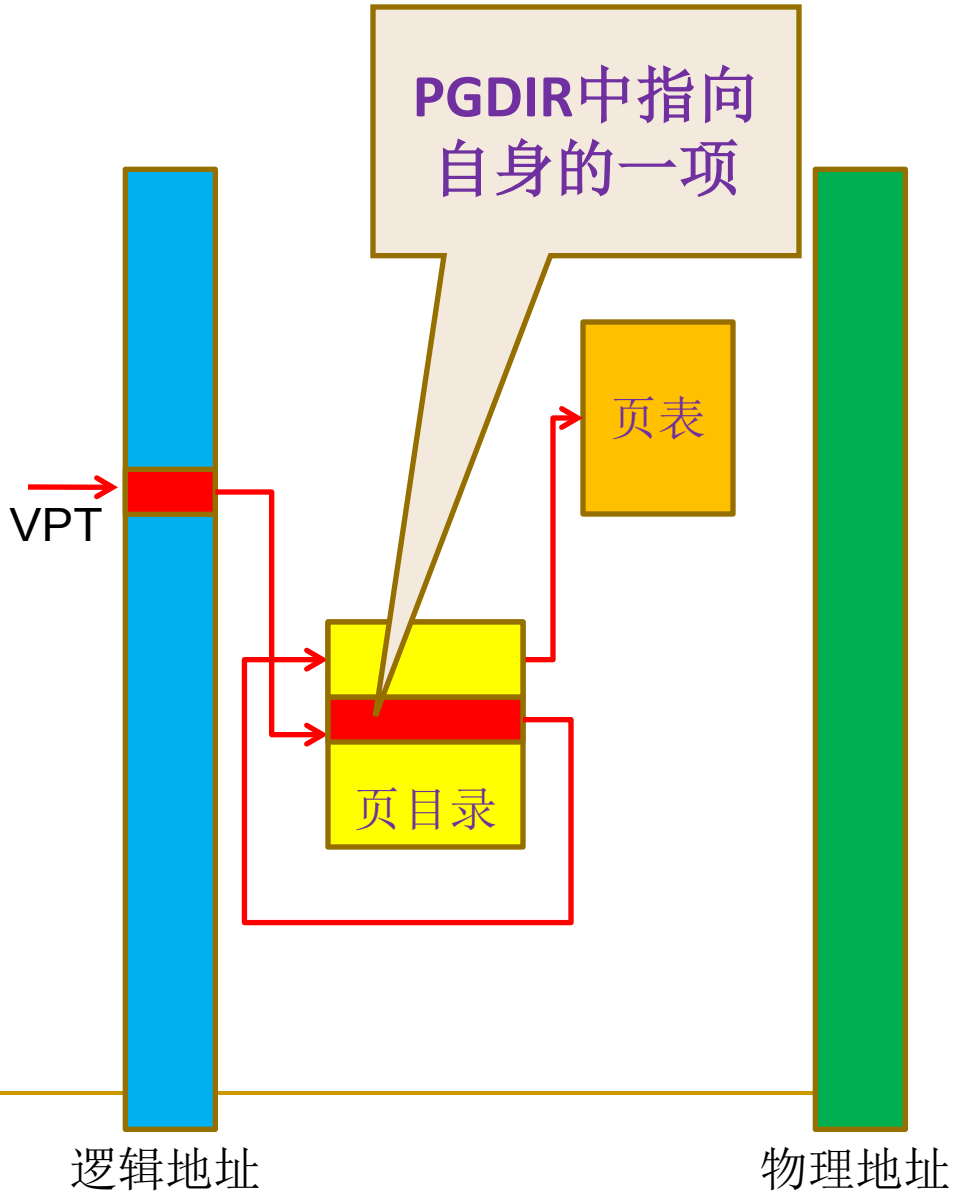
Virtual Page Table 映射

- 非VPT区域内存映射步骤
- 正常的2级页表，三次查找的寻址过程



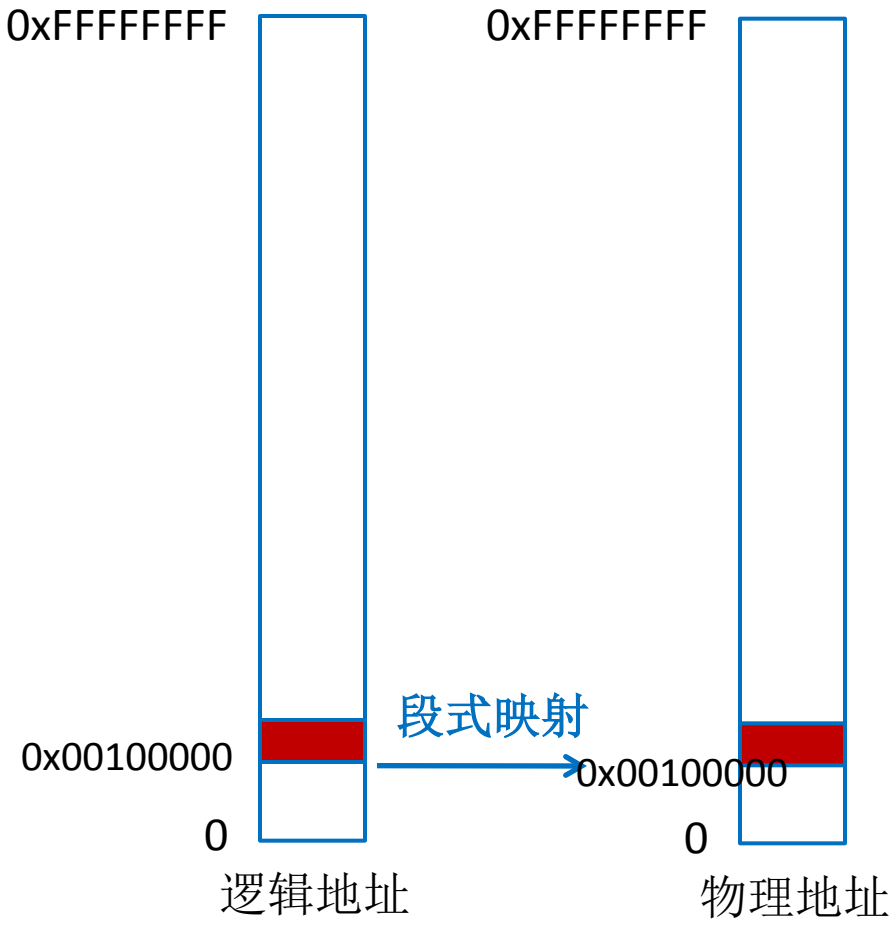
Virtual Page Table 映射

- VPT区域内存映射步骤
- 2级页表的三次查找的寻址过程中，第一次在页目录中转了一圈，故访问到的内存是页表项的内容



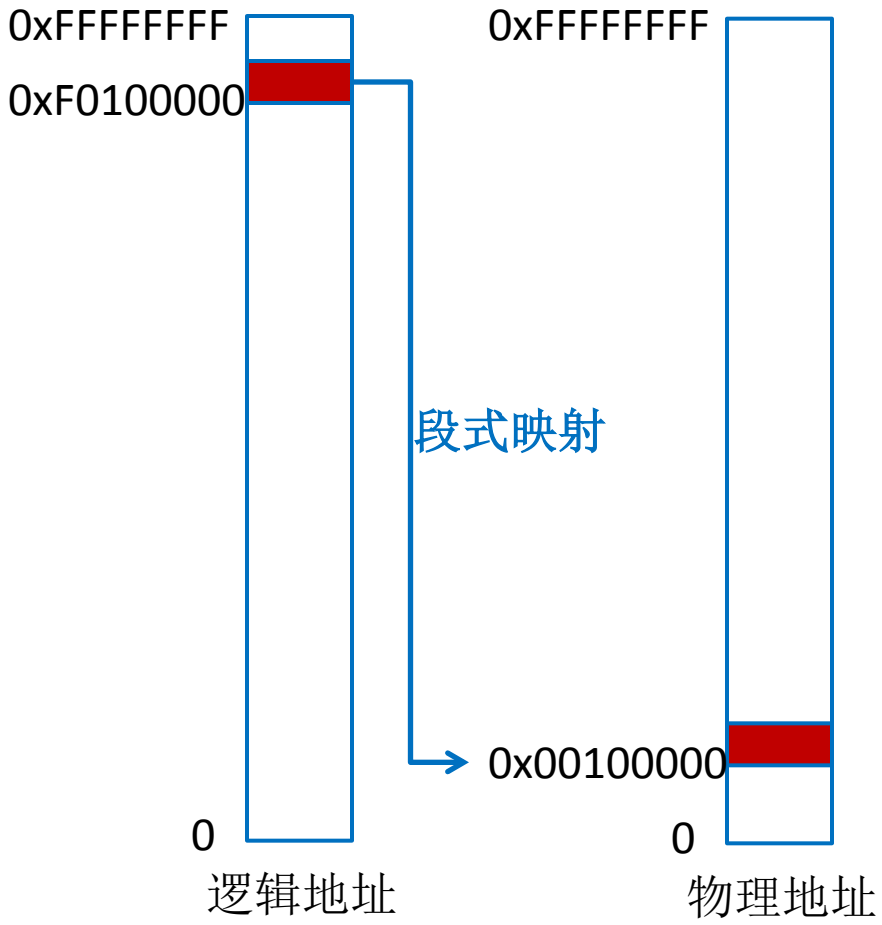
Bootloader运行阶段的地址转换

- BootLoader被加载到0x7c00
  - 逻辑地址=物理地址
- 映射方法
  - 只使用段映射机制
  - GDT的base设为0
- 映射过程
  - X $\xrightarrow{\text{段映射}}$ X



页式尚未开始时的地址转换

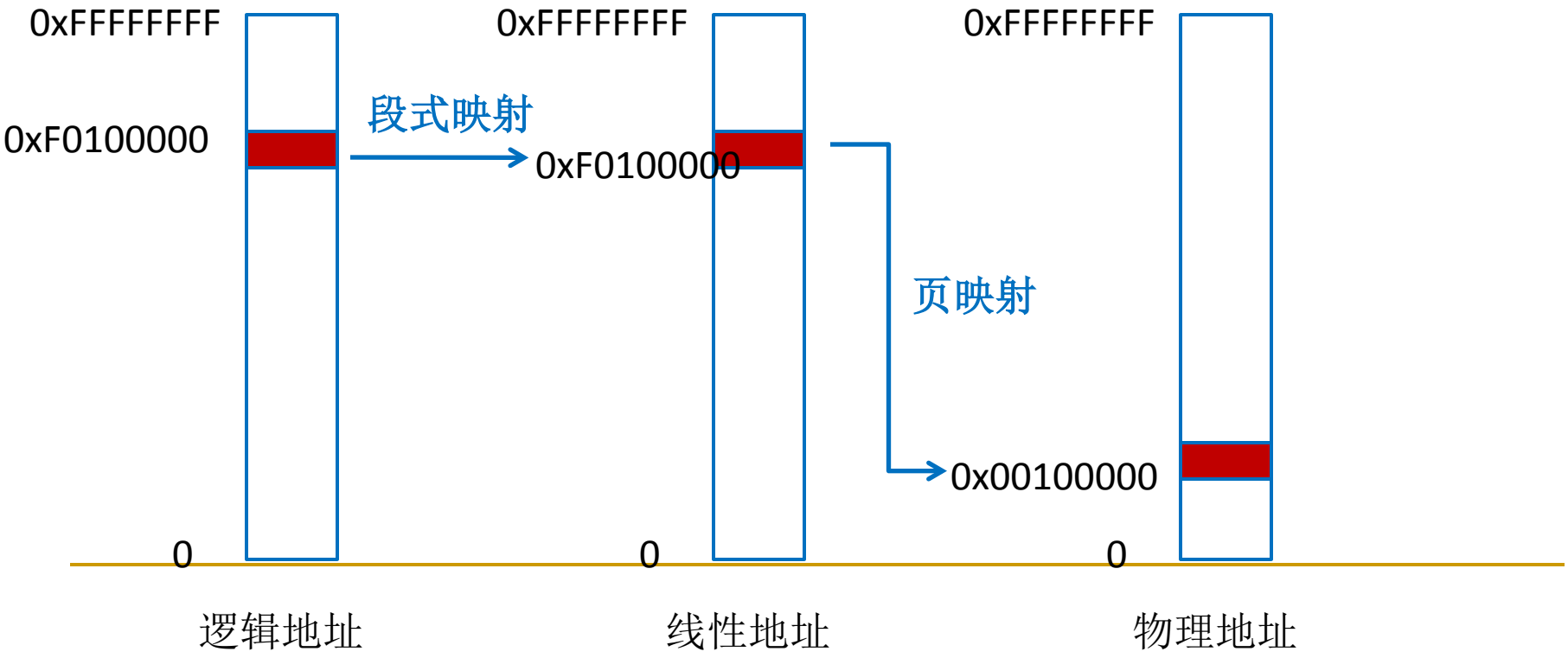
- 内核被加载到物理地址0x00100000
  - 链接地址：0xF0100000
  - 加载地址：0x00100000
- 映射方法
  - 只使用段映射机制
  - Base=-KERNBASE
- 映射过程
  - X + KERNBASE $\xrightarrow{\text{段映射}}$ X



开启页式、段式无效时的地址转换

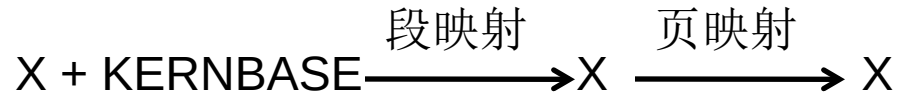
- 内核初始化页表和页目录，打开页映射，设置GDT的base为0，使段式无效
- 此时的映射

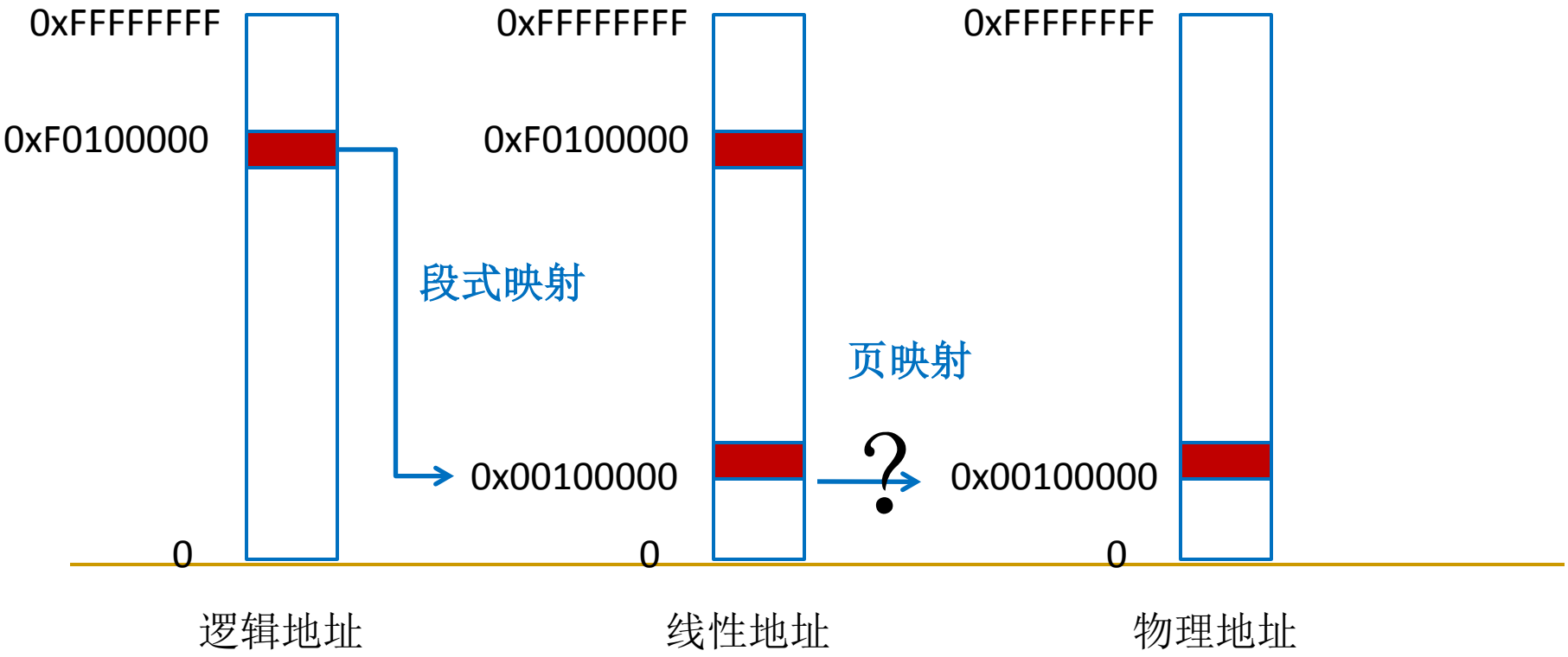
X + KERNBASE —段映射→ X + KERNBASE —页映射→ X



过渡阶段的页目录(1)

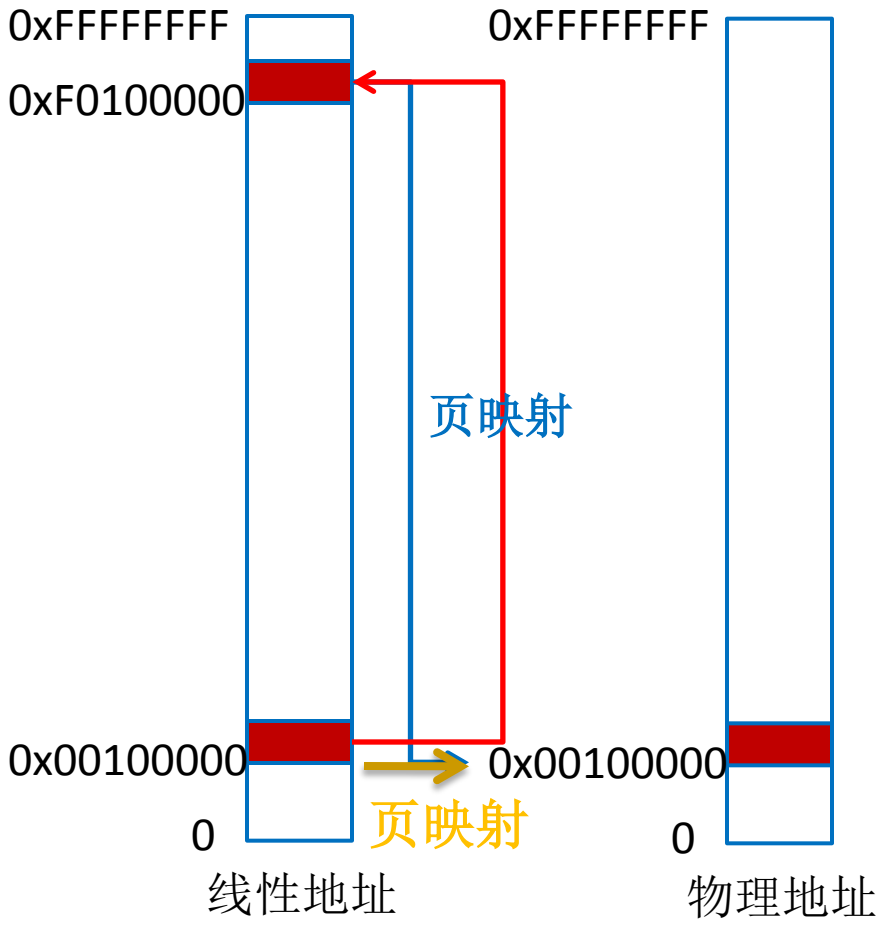
- 内核初始化页表和页目录，打开页映射，此时GDT的base仍然是-KERNBASE
- 此时的映射





过渡阶段的页目录(2)

- 将0xf0100000处的映射复制到0x0处
  - Pgdir[0]=pgdir[PDX(KERNBASE)]
- 将GDT的base设为0后，恢复页目录



参考资料

- elf文件格式：http://os.pku.edu.cn:8080/gaikuang/files/09ospro/readings/elf.pdf
- stabs文档：http://sources.redhat.com/gdb/onlinedocs/stabs_toc.html（local copy on os.pku.edu.cn）
- Intel手册 http://os.pku.edu.cn:8080/gaikuang/files/09ospro/readings/ia32/IA32-3A.pdf

Q&A