

# Lab2: 内存管理

# JOS如何获知物理内存大小？ 如何管理物理内存？

- ▶ 使用i386\_detect\_memory函数获知物理内存大小
  - ▶ 通过nvram\_read从mc146818芯片读取
- ▶ 使用pages数组来存储物理页的状态
- ▶ 使用page\_free\_list链表来管理空闲的物理页
- ▶ page\_alloc函数来分配物理页

```
static void
i386_detect_memory(void)
{
    size_t npages_extmem;

    // Use CMOS calls to measure available base & extended memory.
    // (CMOS calls return results in kilobytes.)
    npages_basemem = (nvram_read(NVRAM_BASELO) * 1024) / PGSIZE;
    npages_extmem = (nvram_read(NVRAM_EXTLO) * 1024) / PGSIZE;

    // Calculate the number of physical pages available in both base
    // and extended memory.
    if (npages_extmem)
        npages = (EXTPHYSMEM / PGSIZE) + npages_extmem;
    else
        npages = npages_basemem;

    cprintf("Physical memory: %uK available, base = %uK, extended = %uK\n",
        npages * PGSIZE / 1024,
        npages_basemem * PGSIZE / 1024,
        npages_extmem * PGSIZE / 1024);
}
```

# 页表建立之前物理内存的分配方式是什么，分配了哪些物理内存？

- ▶ 使用boot\_alloc函数，根据entrypgdir.c中的硬编码页表分配，顺序地找到下一个空闲页
  - ▶ boot\_alloc维护了nextfree指针，按顺序指向下一个空闲页
- ▶ 使用boot\_alloc分配的空间：页目录、存储物理页表链表的空间

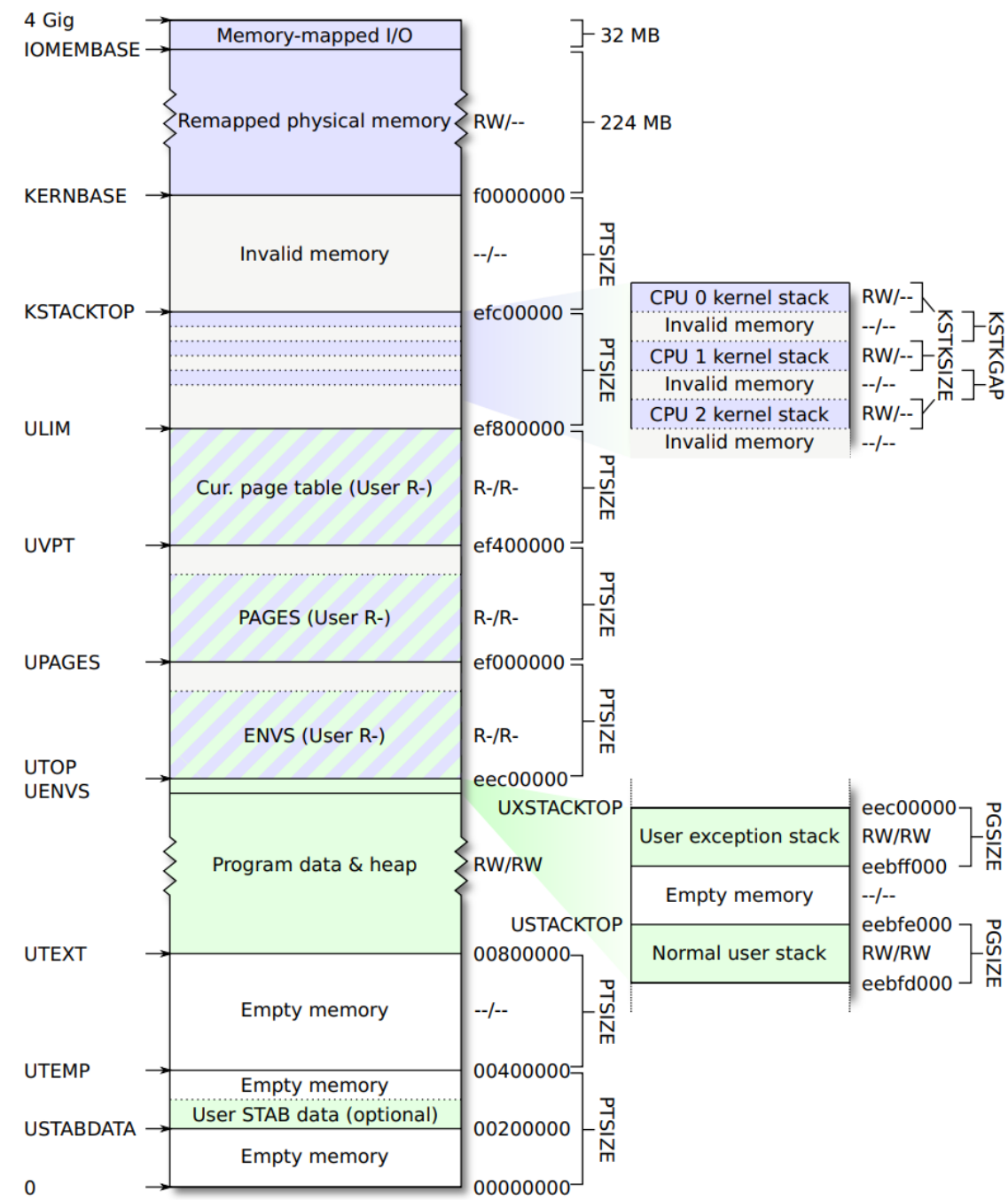
# 启用分页和跳转到高地址之间为什么能够继续在低地址运行

- ▶ 在内核开始时，加载的页面目录是entry\_pgdir，它将虚拟地址[0, 4M)和[KERNBASE, KERNBASE + 4MB)映射到物理地址[0, 4M)，因此内核正常运行。
- ▶ 但是在函数mem\_init()中，加载了kern\_pgdir，它将低虚拟地址[0, 4M)映射取消了，因此有必要跳转到KERNBASE之上的高EIP。

```
pde_t entry_pgdir[NPDENTRIES] = {  
    // Map VA's [0, 4MB) to PA's [0, 4MB)  
    [0]  
        = ((uintptr_t)entry_pgtable - KERNBASE) + PTE_P,  
    // Map VA's [KERNBASE, KERNBASE+4MB) to PA's [0, 4MB)  
    [KERNBASE>>PDXSHIFT]  
        = ((uintptr_t)entry_pgtable - KERNBASE) + PTE_P + PTE_W  
};
```

```
__attribute__((__aligned__(PGSIZE)))  
pte_t entry_pgtable[NPTENTRIES] = {  
    0x000000 | PTE_P | PTE_W,  
    0x001000 | PTE_P | PTE_W,  
    0x002000 | PTE_P | PTE_W,  
    0x003000 | PTE_P | PTE_W,  
    0x004000 | PTE_P | PTE_W,  
    0x005000 | PTE_P | PTE_W,  
    0x006000 | PTE_P | PTE_W,  
    0x007000 | PTE_P | PTE_W,  
    0x008000 | PTE_P | PTE_W,  
    0x009000 | PTE_P | PTE_W,  
    0x00a000 | PTE_P | PTE_W,  
    0x00b000 | PTE_P | PTE_W,  
    0x00c000 | PTE_P | PTE_W,  
    0x00d000 | PTE_P | PTE_W,  
    0x00e000 | PTE_P | PTE_W,  
    0x00f000 | PTE_P | PTE_W,  
    0x010000 | PTE_P | PTE_W,  
    0x011000 | PTE_P | PTE_W,  
    0x012000 | PTE_P | PTE_W,  
    0x013000 | PTE_P | PTE_W,  
    0x014000 | PTE_P | PTE_W,  
    0x015000 | PTE_P | PTE_W,  
    0x016000 | PTE_P | PTE_W,  
    0x017000 | PTE_P | PTE_W,
```

# JOS Virtual Memory Map



## mem\_init之后的虚拟空间布局?

```
boot_map_region(kern_pgdir, UPAGES, PTSIZE, PADDR(pages), PTE_U | PTE_P);

////////////////////////////////////
// Use the physical memory that 'bootstack' refers to as the kernel
// stack. The kernel stack grows down from virtual address KSTACKTOP.
// We consider the entire range from [KSTACKTOP-PTSIZE, KSTACKTOP)
// to be the kernel stack, but break this into two pieces:
// * [KSTACKTOP-KSTKSIZE, KSTACKTOP) -- backed by physical memory
// * [KSTACKTOP-PTSIZE, KSTACKTOP-KSTKSIZE) -- not backed; so if
// the kernel overflows its stack, it will fault rather than
// overwrite memory. Known as a "guard page".
// Permissions: kernel RW, user NONE
// Your code goes here:
boot_map_region(kern_pgdir, KSTACKTOP - KSTKSIZE, KSTKSIZE, PADDR(bootstack), PTE_W);

////////////////////////////////////
// Map all of physical memory at KERNBASE.
// Ie. the VA range [KERNBASE, 2^32) should map to
// the PA range [0, 2^32 - KERNBASE)
// We might not have 2^32 - KERNBASE bytes of physical memory, but
// we just set up the mapping anyway.
// Permissions: kernel RW, user NONE
// Your code goes here:
boot_map_region(kern_pgdir, KERNBASE, 0xffffffff - KERNBASE + 1, 0, PTE_W);
```

# JOS最大可以管理多少物理内存?

- ▶ 利用一个大小为4MB的空间UPAGES来存放所有的页的PageInfo结构体信息，每个结构体的大小为8B，所以一共可以存放512K个PageInfo结构体，所以一共可以出现512K个物理页，每个物理页大小为4KB，自然总的物理内存占2GB。

## 管理最大数量的物理内存需要多少空间开销？

- ▶ 从上面知道，总共可以使用512K页，也就是说有512K PTES，因为一页可以存储1K PTES，所以需要512页 ( $512 \times 4\text{KB} = 2\text{MB}$ ) 来存储整个PTES。
- ▶ 另外需要一页来存储4KB的页目录。这里我们得出内存管理的空间开销是 $4\text{MB}(\text{pages}) + 2\text{MB}(\text{page tables}) + 4\text{KB}(\text{page directory})$ ，总共6MB+4KB。

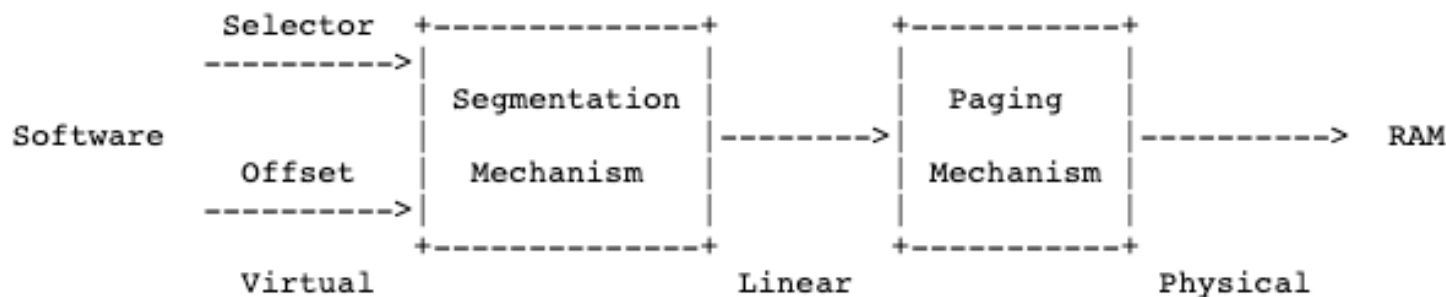
# 加载地址、链接地址、虚拟地址、物理地址

- ▶ 加载地址和链接地址都可以既是物理地址又是虚拟地址吗？
- ▶ Bootloader的加载地址是物理地址还是虚拟地址？
- ▶ 内核的加载地址、链接地址分别是虚拟地址还是物理地址？
- ▶ 取决于有没有启动分页。
- ▶ Bootloader加载时页表没有建立，加载地址是物理地址。
- ▶ Kernel的加载地址是物理地址，链接地址是虚拟地址。
- ▶ 后续的用户进程，加载地址和链接地址都是虚拟地址。

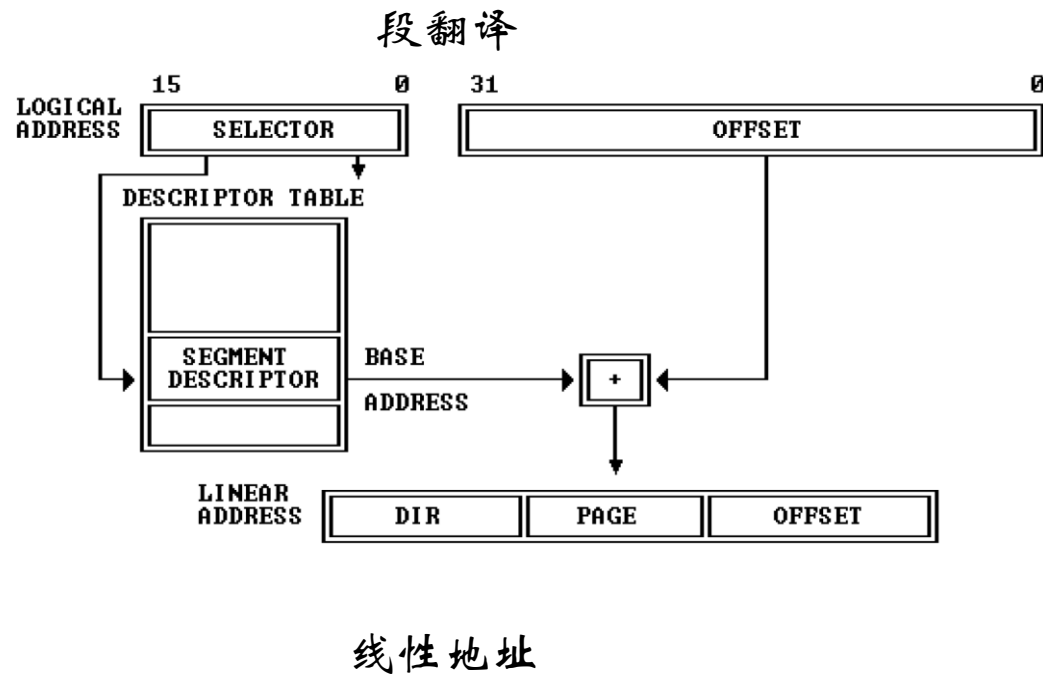


# 虚拟地址、逻辑地址、线性地址和物理地址的关系

- ▶ 虚拟地址：段选择符和段内偏移 (selector:offset)
- ▶ 线性地址：虚拟地址经过段描述符翻译得到的地址
- ▶ 物理地址：线性地址经过页表翻译得到的地址，是最终地址总线上的值，也是访问位置在RAM中的实际位置
- ▶ 逻辑地址？
  - ▶ 逻辑地址和虚拟地址表达的是同一种东西，都是编译器产生的地址
  - ▶ 不过两者在语义上有轻微的差别，逻辑地址是和线性地址作为对立的一对概念；而虚拟地址则是相对物理地址而言的。



# 从虚拟地址到物理地址的转换过程



页翻译

Figure 5-9. Page Translation

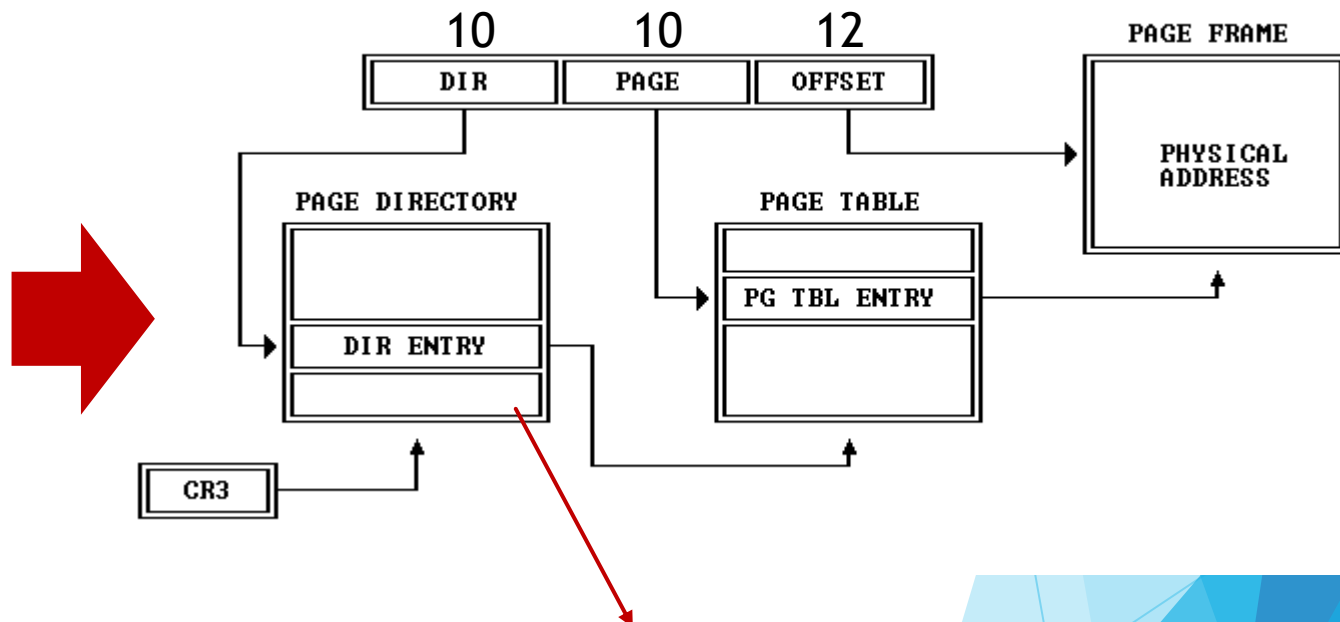
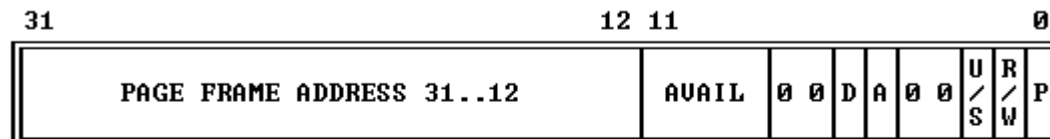


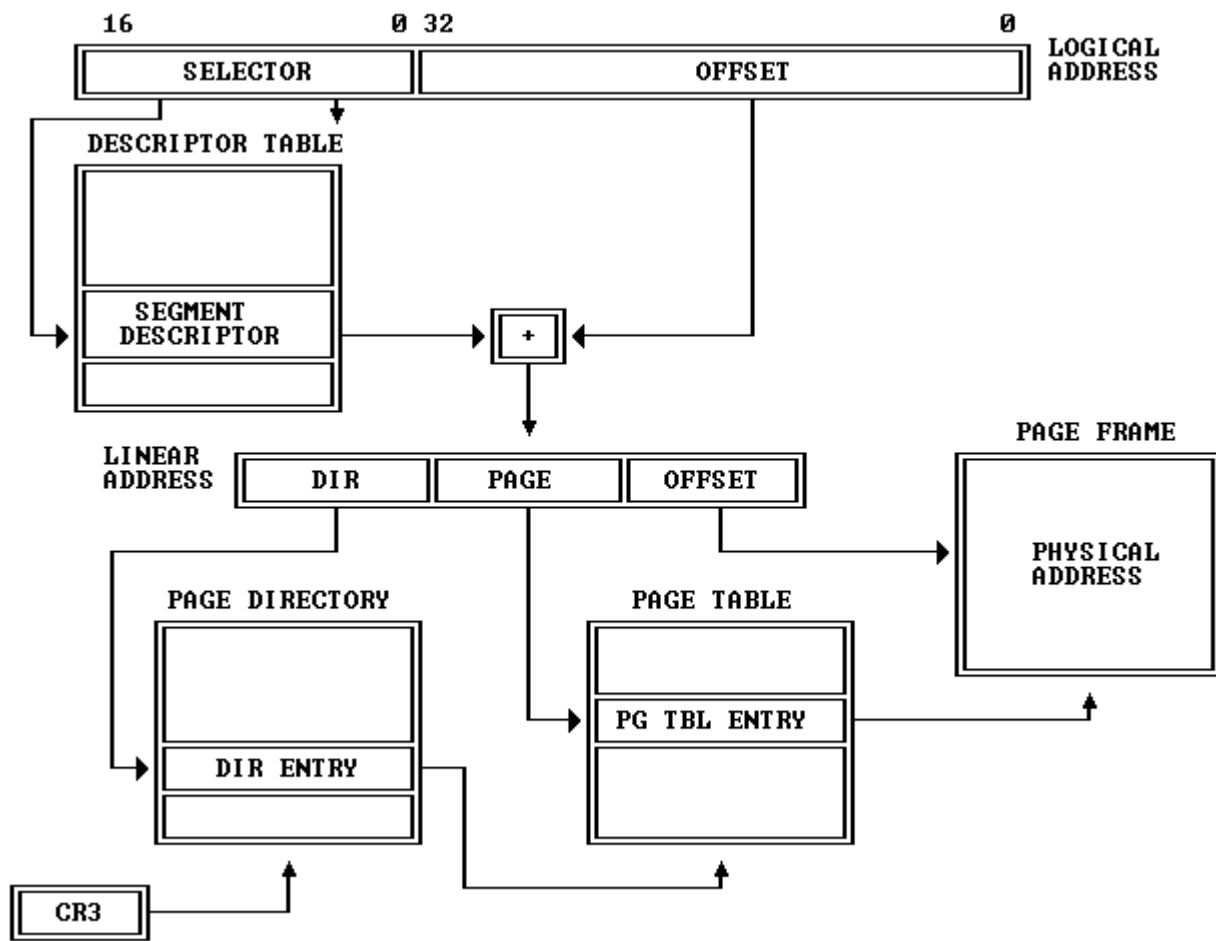
Figure 5-10. Format of a Page Table Entry



- P - PRESENT
  - R/W - READ/WRITE
  - U/S - USER/SUPERVISOR
  - D - DIRTY
  - AVAIL - AVAILABLE FOR SYSTEMS PROGRAMMER USE
- NOTE: 0 INDICATES INTEL RESERVED. DO NOT DEFINE.

# 从虚拟地址到物理地址的转换过程

Figure 5-12. 80386 Addressing Mechanism



# JOS用户程序为什么不能读写内核内存

- ▶ 使用 USER 位来限制用户应用程序的访问。
- ▶ 如果 USER 位清零，则该页面仅可供内核访问。
- ▶ 用户的 CPL 为 3，只有 0-2 之间的 CPL 才能访问没有设置 USER 位的页面，因此内核内存受到保护。

# 内存管理中的保护机制

## ▶ 保护模式

- ▶ 在保护模式下，用户进程不能随意访问物理内存
  - ▶ 用户态必须通过页表翻译，页表受到内核态的管理

## ▶ 分段管理

- ▶ 通过段描述符来确定地址基值（用户无权直接读取段描述符）
- ▶ 不同的段按照段中数据的重要性和代码的可信程度，给定不同的特权级别
- ▶ 每当一个程序试图访问一个段时，就把CPL与要访问的段的特权级进行比较，以决定是否允许进行访问

## ▶ 实现了不同进程之间的保护

- ▶ 用户进程使用不同的页表，不能直接修改其他进程占用的内存页

## ▶ 栈的特殊保护

- ▶ 设置金丝雀值等以防止栈中保存的值被修改

# 页表自映射

## ▶ 页表翻译

- ▶ 页目录（一级）、页表（二级）没有虚拟地址
- ▶ 需要使用虚拟地址访问页表
  - ▶ 查询运行时内存占用
  - ▶ 查询页面权限
  - ▶ ...

## ▶ 页表加入虚拟地址空间

- ▶ 页表内需要包含【页表自身VA=>PA的映射】
- ▶ 页目录的某一条entry，其PFN指向了页目录自己

怎么在一级页表（作为一级页表）里记录

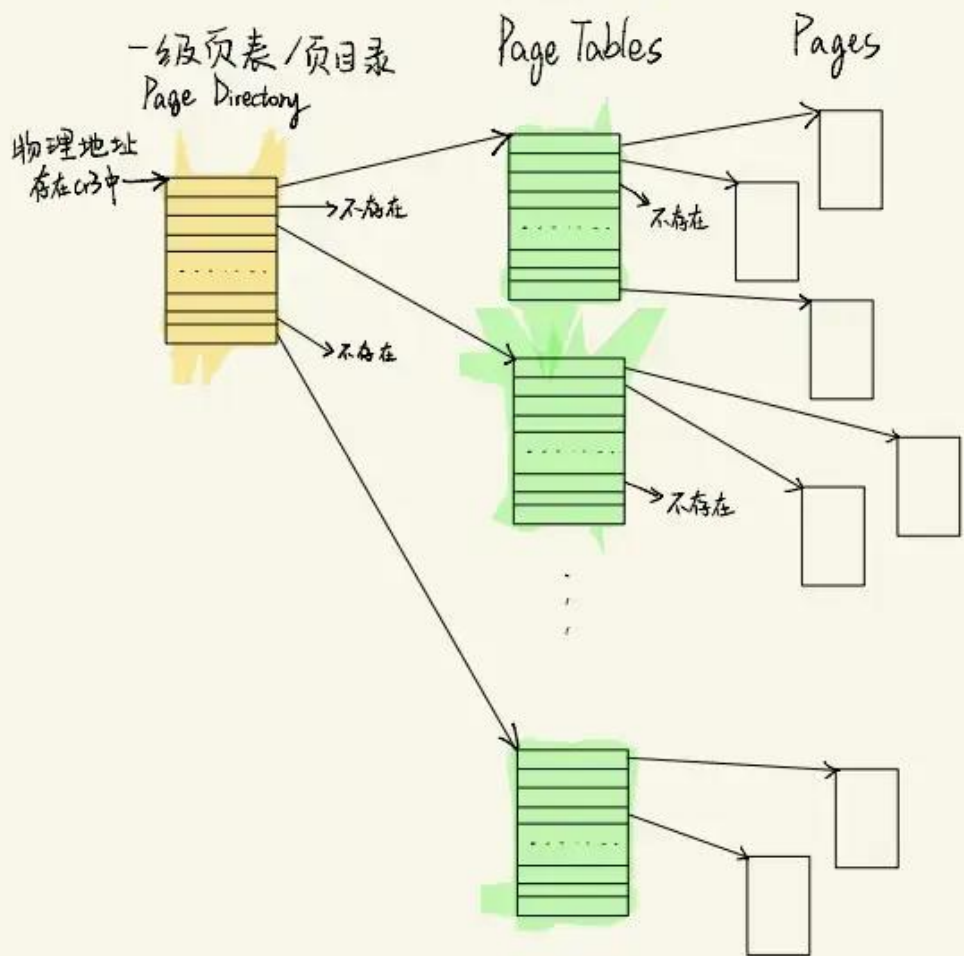
一级页表（作为二级页表们的“二级页表”）？



知乎 @涵某

混沌状态

物理空间



是物理页  
没有规定虚拟地址

有虚拟地址

根据虚拟地址可以求出  
物理地址

知乎 @涵某

# 页表自映射

假设页目录的第 $t$ 项指向了页目录自己

$$PD[t].PFN = PFN(PD)$$

此时页目录作为二级页表进行翻译

页目录的每一项都应当指向页表的物理页框

$$PD[i].PFN = PFN(PT[i])$$

第 $i$ 个页表 $PT[i]$ 的

$$\text{虚拟页框号 } VFN(PT[i]) = t \mid i \mid 00B$$

$$\text{虚拟地址 } VA(PT[i]) = t \mid i \mid 000H$$

其中， $PT[t]$ 是 $PD$

**$t$  为基地址的高10位：957**

**可以直接在JOS中打印出来看**

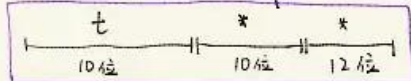
思考1：JOS将页表放在了 $UVPT=0xef400000$ 区域中，对应 $t$ 是多少？

思考2：JOS中是如何体现页表自映射的？

```
// Permissions: kernel R, user R
kern_pgdir[PDX(UVPT)] = PADDR(kern_pgdir) | PTE_U | PTE_P;
```

reference: 《什么是“页表自映射”？》 <https://zhuanlan.zhihu.com/p/452598045>

不如把 1024 个 (随便放哪的) “二级页表”

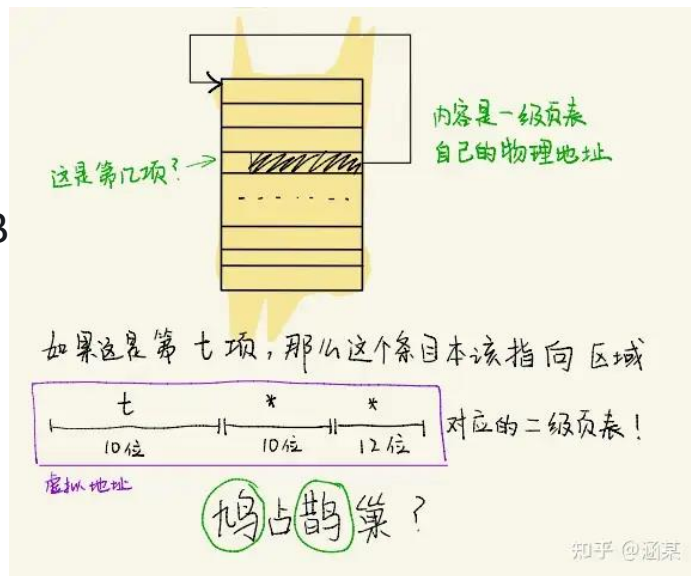
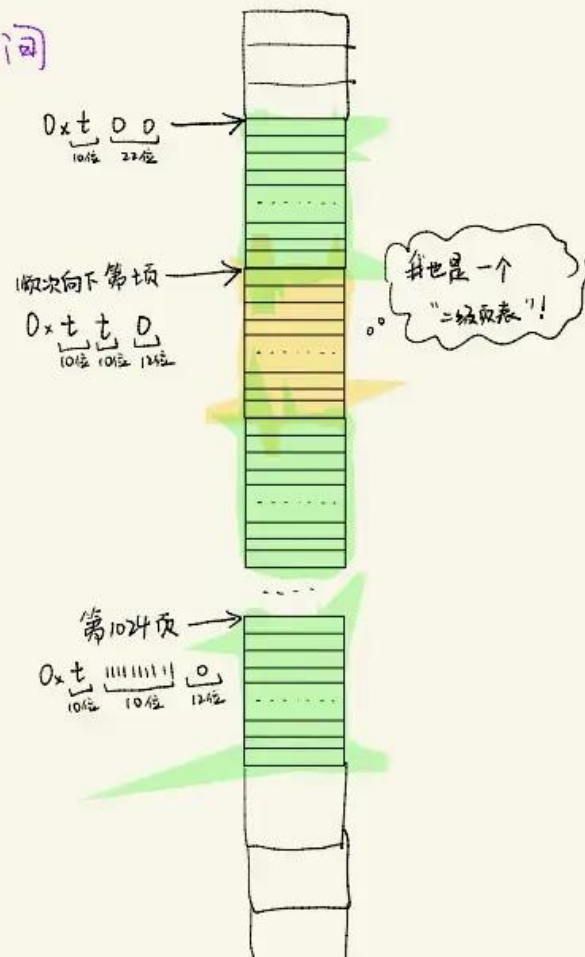
虚拟地放在  这片虚拟空间！

韵 := 鸠 ✓

现在，所有“二级页表”都要放在  $0xt00 \sim 0xt00 + 4MB$

范围内，第 $t$ 个“二级页表”——一级页表也不例外。

虚拟空间



知乎 @涵某

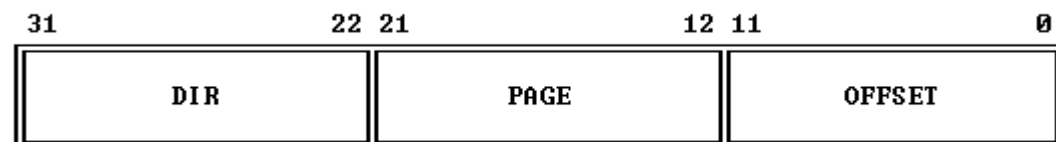
知乎 @涵某



# 用户进程如何获得一个虚拟地址所在物理页面的地址以及二级页表的地址？

- ▶ 假设要查询的虚拟地址是  $vaddr = DIR \mid PAGE \mid OFFSET$  (10 | 10 | 12)
  - ▶ UVPT 4MB对齐, 后22位为全0 (JOS中UVPT=0xef400000)
  - ▶ 二级页表PT的虚拟地址  $VA(PT) = UVPT + DIR * 4K = \underline{UVPT[31:22] \mid DIR \mid 000H}$
  - ▶ PTE的虚拟地址  $VA(PTE) = UVPT[31:22] \mid DIR \mid PAGE \mid 00B$
  - ▶  $PA(vaddr) = PTE.PFN \mid OFFSET$
- ▶ 二级页表的物理地址:
  - ▶  $vaddr = VA(PT) = UVPT[31:22] \mid DIR \mid 000H$ , 求  $PA(PT)$
  - ▶  $DIR_{PT} = UVPT[31:22]$ ,  $PAGE_{PT} = DIR$ ,  $OFFSET_{PT} = 000H$
  - ▶  $VA(PTE) = UVPT[31:22] \mid UVPT[31:22] \mid DIR \mid 00B$ , 然后对PTE解引用获得二级页表PFN

Figure 5-8. Format of a Linear Address





# Inc/assert.h中的宏为什么要写成只执行一次的循环?

```
#define assert(x) \
    do { if (!(x)) panic("assertion failed: %s", #x); } while (0)
```

## ■ 程序语句:

```
if (condition)
    foo;
else
    bar;
```

## ■ 宏定义:

```
#define foo \
    do { \
        if (condition) \
            statement; \
    } while (0);
```

## ■ 宏替换之后:

```
if (condition)
    do {
        if (condition)
            statement;
    } while (0);
else
    bar
```



# Inc/assert.h中的宏为什么要写成只执行一次的循环?

## ■ 程序语句:

```
if (condition)
    foo;
else
    bar;
```

## ■ 宏定义:

```
#define foo \
    if (condition) \
        statement
```

## ■ 宏替换之后:

```
if (condition)
    if (condition)
        statement;
else
    bar;
```



## Inc/assert.h中的宏为什么要写成只执行一次的循环?

■ 程序语句:

```
if (condition)
    foo;
else
    bar;
```

■ 宏定义:

```
#define foo \
    {state1; \
      state2; }
```

■ 宏替换之后:

```
if (condition)
{
    state1;
    state2;
};
else
    bar;
```



Thanks