# 计算机系统基础

# 实验报告

# PA 3

计算机科学与技术系
191220008 陈南曈

## 3-3：

### 1、Kernel 的虚拟页和物理页的映射关系是什么？请画图说明。

在 memory.h 等文件中有如下宏定义：

```
/* 32bit x86 uses 4KB page size */
#define PAGE_SIZE 4096
#define NR_PDE 1024
#define NR_PTE 1024
#define PAGE_MASK (4096 - 1)
#define PT_SIZE ((NR_PTE) * (PAGE_SIZE))

/* force the data to be aligned with page boundary.
   statically defined page tables uses this feature. */
#define align_to_page __attribute((aligned(PAGE_SIZE)))


/* the maxinum loader size is 16MB */
#define KMEM (16 * 1024 * 1024)

/* NEMU has 128MB physical memory  */
#define PHY_MEM (128 * 1024 * 1024)

#define make_invalid_pde() 0
#define make_invalid_pte() 0
#define make_pde(addr) ((((uint32_t)(addr)) & 0xfffff000) | 0x7)
#define make_pte(addr) ((((uint32_t)(addr)) & 0xfffff000) | 0x7)
```

对 kvm.c 进行查看，发现这段代码对 kernel 的页目录和页表进行了初始化：

```
/* make all PDE invalid */
memset(pdir, 0, NR_PDE * sizeof(PDE));

/* fill PDEs and PTEs */
pframe_idx = 0;
for (pdir_idx = 0; pdir_idx < PHY_MEM / PT_SIZE; pdir_idx++)
{
    pdir[pdir_idx].val = make_pde(ptable);
    pdir[pdir_idx + KOFFSET / PT_SIZE].val = make_pde(ptable);
    for (ptable_idx = 0; ptable_idx < NR_PTE; ptable_idx++)
    {
        ptable->val = make_pte(pframe_idx << 12);
        pframe_idx++;
        ptable++;
    }
}
```

由此可知 PHY_MEM / PT_SIZE= 2^27/2^22 = 32， KOFFSET / PT_SIZE= 0xC0000000 /2^ 22 = 0x300，NR_PTE = 2^10。

已知 make_pde 相关宏定义如下：

```
#define make_pde (addr) ((((uint32_t)(addr)) & 0xfffff000) | 0x7)
```

它使这个宏的值的低 3 位都为 1，中间 9 位都为 0，高 20 位保留其参数原来的值。所以只有当 ptable 的高 20 位改变时，make_pde(ptable) 的值才会改变。

对于内层的 for 循环，每次循环 ptable++。每一次外层循环中，内层循环进行 2^10 次循环，ptable 一共增加 2^10。因为 ptable 是按 2^12 对齐的，而每次外层循环中，又有

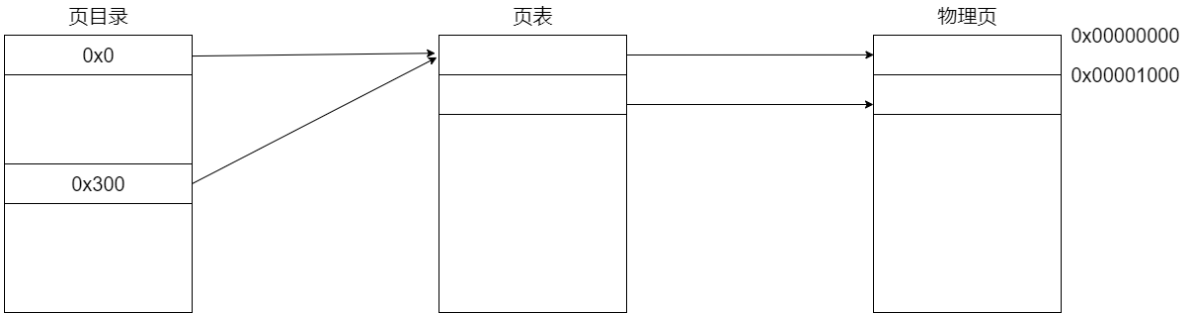
```
pdir [ pdir_idx ] . val = make_pde ( ptable ) ;
pdir [ pdir_idx + KOFFSET / PT_SIZE ] . val = make_pde ( ptable ) ;
```

所以页目录表的的第 pdir_idx 项和第 pdir_idx + 0x300 项都被映射到同一个页表。

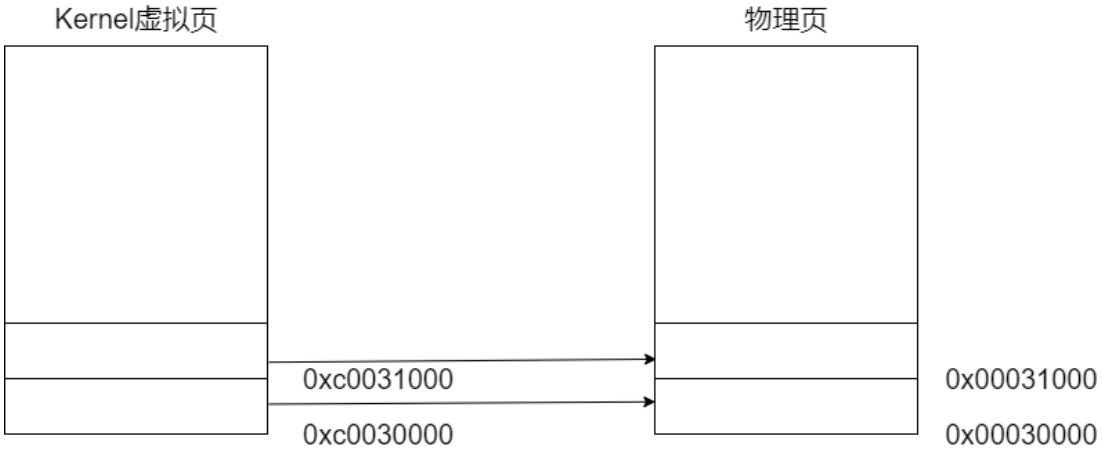
至于虚拟页到物理页的映射，则用下列语句实现 ：

```
ptable ->val = make_pte ( pframe_idx << 12 ) ;
pframe_idx++;
ptable++;
```

pframe_idx 被初始化为 0，左移 12 位之后得到 0x00000000，为第 0 个物理页的首地址；pframe_idx++后为 1，左移 12 位之后得到 0x00001000，为第 1 个物理页的首地址，画图后如下所示：




而 Kernel 的起始地址为 0xc0030000，故 Kernel 的页目录索引从 0x300 开始，在该页目录项对应的页表中，页表项索引从 0x30 开始。故 Kernel 对应的物理页起始地址从 0x00030000 开始，以此类推。



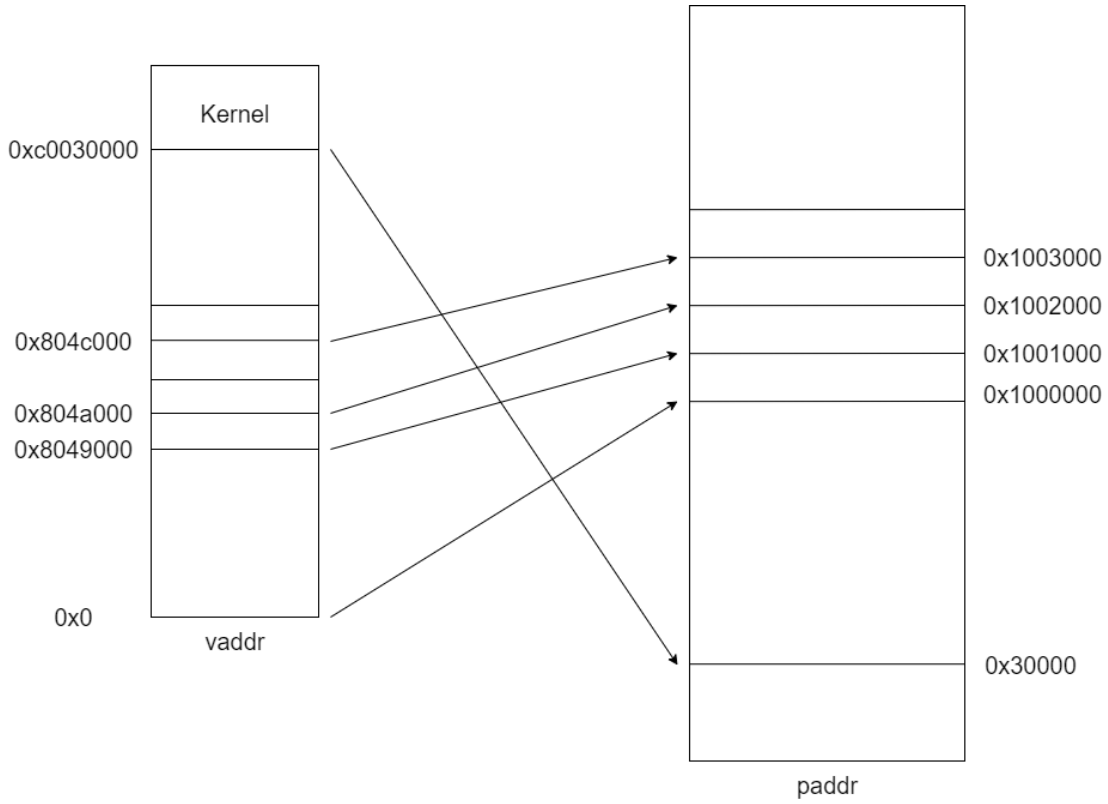

## 2、以某一个测试用例为例，画图说明用户进程的虚拟页和物理页间映射关系又是怎样的？Kernel 映射为哪一段？你可以在 loader()中通过 Log()输出 mm_malloc 的结果来查看映射关系，并结合 init_mm()中的代码绘出内核映射关系。

以 mov.c 为例：

在 loader() 中通过 Log() 输出 mm_malloc 的结果来查看映射关系，得到的结果如下：

```
nemu trap output: [src/main.c,82,init_cond] {kernel} Hello, NEMU world!
nemu trap output: [src/elf/elf.c,29,loader] {kernel} ELF loading from ram disk.
nemu trap output: [src/elf/elf.c,48,loader] {kernel} vaddr = 0x0, paddr = 0x1000000
nemu trap output: [src/elf/elf.c,48,loader] {kernel} vaddr = 0x8049000, paddr = 0x1001000
nemu trap output: [src/elf/elf.c,48,loader] {kernel} vaddr = 0x804a000, paddr = 0x1002000
nemu trap output: [src/elf/elf.c,48,loader] {kernel} vaddr = 0x804c000, paddr = 0x1003000
```

由第一问可知，kernel 被映射到物理地址为 0x30000 及以后一部分的地址。内核映射关系如图所示：




## 3、“在 Kernel 完成页表初始化前，程序无法访问全局变量”这一表述是否正确？在 init_page()里面我们对全局变量进行了怎样的处理？

这句表述不正确，因为我们可以通过 va_to_pa 宏在页表初始化之前直接通过逻辑地址访问到全局变量。

在 init_page() 里面我们对全局变量进行了按页对齐的处理。