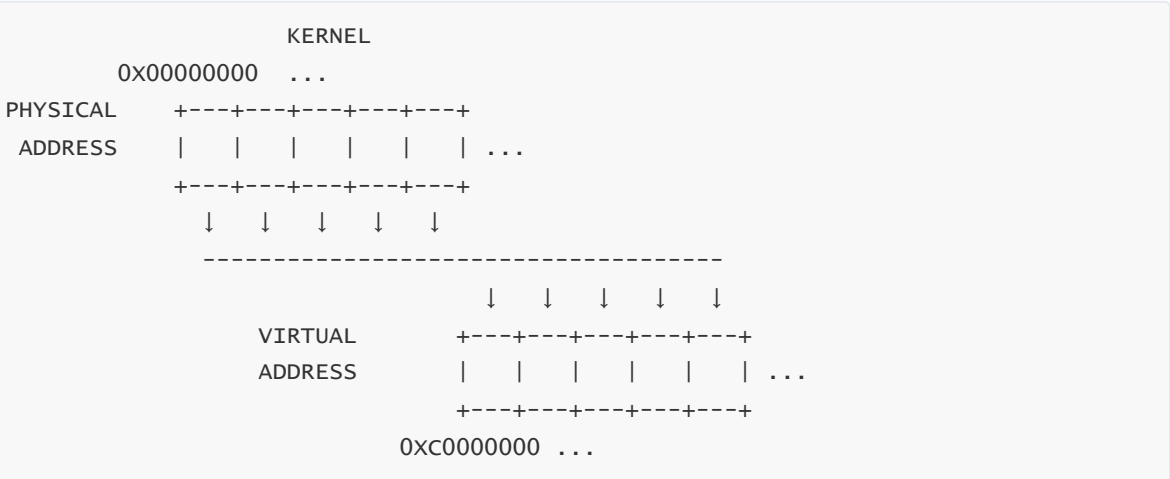
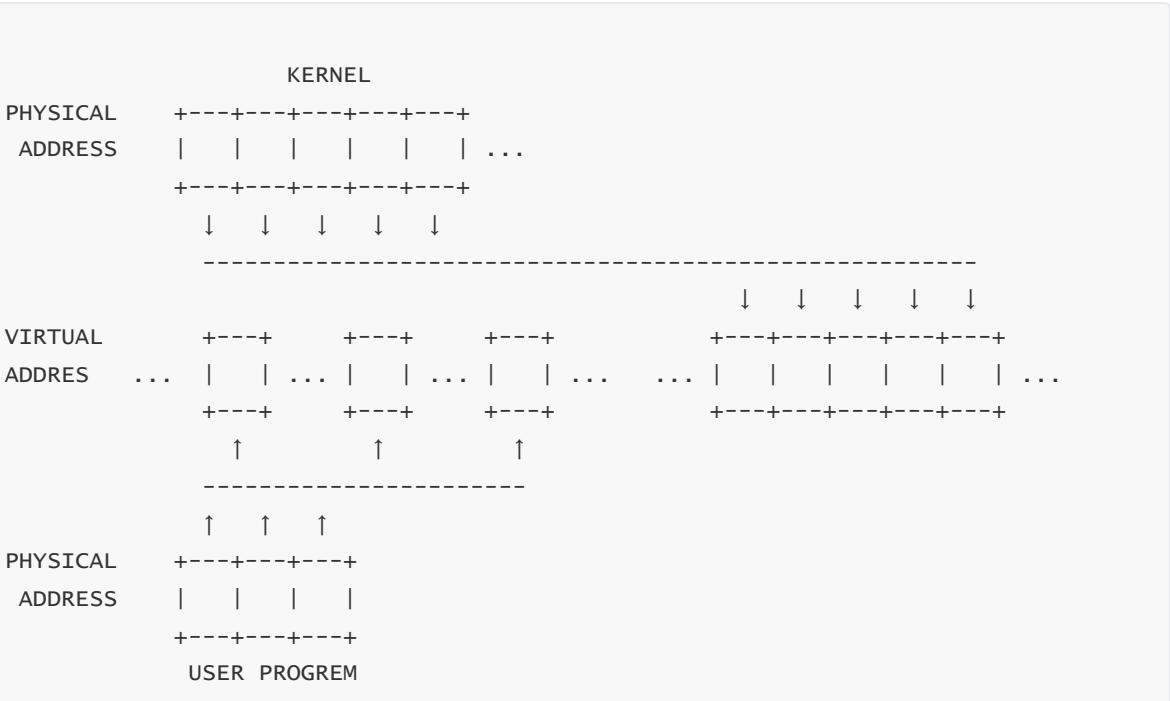
Kernel的虚拟页和物理页的映射关系是什么？请画图说明；

Kernel的代码和数据被映射到虚拟地址空间的高地址部分， 即`0xc0000000 ~ 0xffffffff`。这样的映射关系确保了内核的代码和数据在逻辑上是连续的，方便操作系统的执行和管理。



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。



/kernel/src/memory/mm.c

```
void init_mm()
{
    PDE *kpdir = get_kpdir();
    /* make all PDE invalid */  //清空用户空间的页目录表
    memset(updir, 0, NR_PDE * sizeof(PDE));
    /* create the same mapping above 0xc0000000 as the kernel mapping does */
    //将内核地址空间从0xC0000000到0xFFFFFFFF映射到用户进程的地址空间中
    memcpy(&updir[KOFFSET / PT_SIZE], &kpdir[KOFFSET / PT_SIZE],
           (PHY_MEM / PT_SIZE) * sizeof(PDE));
    ucr3.val = (uint32_t)va_to_pa((uint32_t)updir) & ~0x3ff;
}
```

`init_mm()` 函数的主要目的是初始化用户空间的页目录表，并与内核空间的页目录表建立共享映射。这样可以确保用户进程在访问地址空间时，可以共享内核的部分代码和数据，从而提高系统的效率。

“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？

正确。当 Kernel 完成页表初始化后，进程的虚拟地址空间和物理地址空间之间的映射关系才得以建立，程序才能正确地访问全局变量。

`init_page()` 里对全局变量的处理： 见注释

/kernel/src/memory/kvm.c

```
/* set up page tables for kernel */
void init_page(void)
{
    CR0 cr0;
    CR3 cr3;
    PDE *pdir = (PDE *)va_to_pa(kpdir);          //页目录表的物理地址
    PTE *ptable = (PTE *)va_to_pa(kptable);      //页表的物理地址
    uint32_t pdir_idx, ptable_idx, pframe_idx;

    /* make all PDE invalid */ //清空页目录表
    memset(pdir, 0, NR_PDE * sizeof(PDE));

    /* fill PDEs and PTEs */ //填充页目录项和页表项
    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++;
        }
```

```
    }

    /* make CR3 to be the entry of page directory */ //设置CR3寄存器为页目录表基地址
    cr3.val = 0;
    cr3.page_directory_base = ((uint32_t)pdir) >> 12;
    write_cr3(cr3.val);

    /* set PG bit in CR0 to enable paging */ //启用分页机制 CR0的PG为设为1
    cr0.val = read_cr0();
    cr0.paging = 1;
    write_cr0(cr0.val);
}
```

实验过程

在`include/config.h`头文件中定义宏`IA32_PAGE`并`make clean`；

- `include/config.h`

```
#define IA32_PAGE
```

修改Kernel和testcase中`Makefile`的链接选项；

- `kernel/Makefile`

```
- LDFLAGS = -Ttext=0x30000 -m elf_i386 # before page
+ LDFLAGS = -Ttext=0xc0030000 -m elf_i386
```

- `testcase/Makefile`

```
- LDFLAGS := -m elf_i386 -e start -Ttext=0x100000
+ LDFLAGS := -m elf_i386 -e start
```

在`CPU_STATE`中添加`CR3`寄存器；

- `nemu/include/cpu/reg.h`

```
typedef union{
    struct{
        uint32_t dummy: 12;
        uint32_t dir: 20;
    };
    uint32_t val;
}CR3;
```

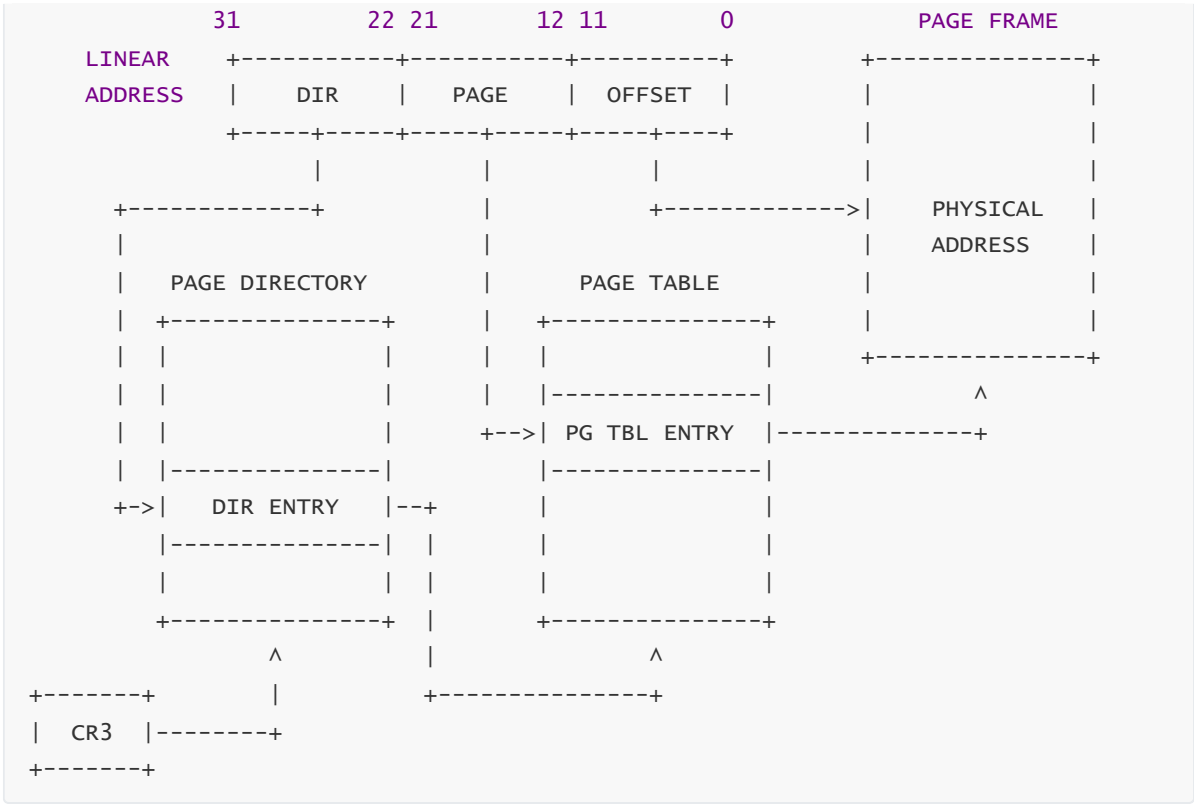
修改`laddr_read()`和`laddr_write()`，适时调用`page_translate()`函数进行地址翻译；

- `nemu/src/memory/memory.c`

```
uint32_t laddr_read(laddr_t laddr, size_t len)
{
    #ifdef IA32_PAGE
        assert(len == 1 || len == 2 || len == 4);
        if(cpu.cr0.pg == 1) {
            if ((laddr & 0xfff) + len > 0x1000) { //data cross the page boundary
                uint8_t data[4] = {0};
                for(int i=0; i<len; i++){
                    data[i] = (uint8_t) laddr_read(laddr+i, 1);
                }
                return *(uint32_t*) data;
            }
            else {
                paddr_t paddr = page_translate(laddr);
                return paddr_read(paddr, len);
            }
        }
        else return paddr_read(laddr, len);
    #else
        return paddr_read(laddr, len);
    #endif
}

void laddr_write(laddr_t laddr, size_t len, uint32_t data)
{
    #ifdef IA32_PAGE
        if(cpu.cr0.pg == 1) {
            if ((laddr & 0xfff) + len > 0x1000) { //data cross the page boundary
                for(int i=0; i<len; i++){
                    laddr_write(laddr+i, 1, ((uint8_t*)&data)[i]);
                }
            }
            else {
                paddr_t paddr = page_translate(laddr);
                paddr_write(laddr, len, data);
            }
        }
        else paddr_write(laddr, len, data);
    #else
        paddr_write(laddr, len, data);
    #endif
}
```

- `nemu/src/memory/mmu/page.c`




```
typedef union{
    struct{
        uint32_t offset: 12;
        uint32_t page: 10;
        uint32_t dir: 10;
    };
    uint32_t val;
}laddr_u;


paddr_t page_translate(laddr_t laddr)
{
#ifndef TLB_ENABLED
    uint32_t paddr = laddr;

    laddr_u la;
    la.val = laddr;

    uint32_t pgdir = cpu.cr3.dir << 12;
    uint32_t pgtbl = paddr_read(pgdir + la.dir*4, 4);
    assert(pgtbl & 1);

    paddr = paddr_read((pgtbl & 0xfffff000) + la.page*4, 4);
    assert(paddr & 1);

    paddr = (paddr & 0xfffff000) + la.offset;

    return paddr;

#else
    return tlb_read(laddr) | (laddr & PAGE_MASK);
#endif
```

```
}
```

修改Kernel的`loader()`，使用`mm_malloc`来完成对用户进程空间的分配；

- `kernel/src/elf/elf.c`

```
#ifdef IA32_PAGE

            uint32_t bg_addr = mm_malloc(va, ph->p_memsz);
            uint32_t pa = (bg_addr & 0xfffff000) + (va & 0xfff);
            va = (uint32_t)(pa_to_va(pa));

#endif
```