Lab2实验报告

> 学号：21371220
>
> 姓名：杨硕

一、思考题

Thinking 2.1

根据指导书，我们实验所使用的R3000 CPU发出的都是虚拟地址，所以

- 在编写的 C 程序中，指针变量中存储的地址是虚拟地址
- 在MIPS 汇编程序中 lw 和 sw 使用的也是虚拟地址

Thinking 2.2

- 对于链表，其创建、插入、遍历、删除等各种操作都较为繁琐，而使用宏来实现链表，好处一是节约了代码量，并且增加了代码可读性；二是提高了代码可重用性，宏本身就是提高代码复用率的手段之一；

  三是使用宏定义链表的操作，如需改动，只需改动宏一处就行，便于修改调整代码
- 假设链表长度为n,链表头为head，对于双向链表、单向链表和循环列表这三种列表：
  1. 插入操作，包括INSERT_HEAD、INSERT_BEFORE、INSERT_AFTER、INSERT_TAIL
  - 对于双向链表:
    - INSERT_HEAD复杂度o(1),
    - INSERT_BEFORE通过prev找到插入位置，复杂度o(1),
    - INSERT_AFTER通过next找到插入位置，复杂度o(1),
    - INSERT_TAIL由于没有专门标记链表结尾的位置，需要遍历到链尾，复杂度o(n);
  - 对于单向链表:
    - INSERT_HEAD复杂度o(1),
    - INSERT_BEFORE无prev找到插入位置，需要从头开始遍历寻找，复杂度为o(n),
    - INSERT_AFTER通过next找到插入位置，复杂度为o(1),
    - INSERT_TAIL由于没有专门标记链表结尾的位置，需要遍历到链尾，复杂度o(n);
  - 对于单向循环链表:
    - INSERT_HEAD复杂度o(1),
    - INSERT_BEFORE无prev找到插入位置，需要从头开始遍历寻找，复杂度为o(n),
    - INSERT_AFTER通过next直接找到插入位置，复杂度为o(1),
    - INSERT_TAIL由于链表的循环特征，可以直接插入，改变head位置即可，复杂度o(1);
  2. 删除操作，LIST_REMOVE
  - 双向链表：

    删除指定元素可以通过prev和next直接定位，复杂度为o(1),

- 单向链表：

  需要遍历寻找删除位置，复杂度o(n)

- 循环链表：

  需要遍历寻找删除位置，复杂度o(n)

综上，双向链表相对于其他两个链表，插入和删除操作的效率更高

Thinking 2.3

```
C
struct Page_list{
    struct {
        struct {
            struct Page *le_next;
            struct Page **le_prev;
        } pp_link;
        u_short pp_ref;
    }* lh_first;
}
```

首先，在`queue.h`中，定义了如下宏：

```
 #define LIST_ENTRY(type)                                        struct {

        struct type *le_next;  /* next element */
        struct type **le_prev; /* address of previous next element */
}
```

在`pmap.h`中又有如下结构体`page`：

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

并且前面声明

```
typedef LIST_ENTRY(Page) Page_LIST_entry_t;
```

那么`page`的完整结构是

```
struct Page {
    struct {
        struct Page *le_next;
        struct Page **le_prev;
    } pp_link;
    u_short pp_ref;
};
```

在`queue.h`中还有另一个宏定义：

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

这里`lh_first`相当于链表头

综上，C的结构是正确的

Thinking 2.4

- 指导书对ASID的作用描述为：

  用于区分不同的地址空间。查找TLB表项时，除了需要提供VPN，还需要提供ASID（同一虚拟地址在不同的地址空间中通常映射到不同的物理地址）。

  网上对ASID的描述为：

  “ASID 唯一标识每个进程并用于为该进程提供地址空间保护。当 TLB 尝试解析虚拟页号时，它会确保当前运行的进程的 ASID 与与虚拟页关联的 ASID 匹配。如果 ASID 不匹配，则将尝试视为 TLB 未命中”。

  即ASID是TLB中每个表项的额外位，用于在当某进程访问该表项时，确认是否是相匹配的进程。这样，TLB就可以同时包含多个进程的表项，在切换进程时，根据ASID识别表项属于哪个进程。

- ASID占用EntryHi6-11位，共6位，所以可容纳不同的地址空间的最大数量为2^6=64

Thinking 2.5

- tlb_invalidate函数内部调用了tlb_out函数
- 调用tlb_out函数，根据ASID和虚拟地址va清空特定的TLB表项
- tlb_out汇编代码如下

```
LEAF(tlb_out)
.set noreorder
    mfc0    t0, CP0_ENTRYHI
    mtc0    a0, CP0_ENTRYHI
    nop
    /* Step 1: Use 'tlbp' to probe TLB entry */
    /* Exercise 2.8: Your code here. (1/2) */
    tlbp
    nop
    /* Step 2: Fetch the probe result from CP0.Index */
    mfc0    t1, CP0_INDEX
.set reorder
    bltz    t1, NO_SUCH_ENTRY
.set noreorder
    mtc0    zero, CP0_ENTRYHI
    mtc0    zero, CP0_ENTRYLO0
    nop
    /* Step 3: Use 'tlbwi' to write CP0.EntryHi/Lo into TLB at CP0.Index  */
    /* Exercise 2.8: Your code here. (2/2) */
    tlbwi
```

```
.set reorder

NO_SUCH_ENTRY:
    mtc0    t0, CP0_ENTRYHI
    j       ra
END(tlb_out)
```

逐行解释：

把CP0_ENTRYHI原有的值写入寄存器t0

将寄存器a0的值保存到CP0_ENTRYHI中

nop等待

根据CP0_ENTRYHI中的 Key（包含 VPN 与 ASID），查找 TLB 中与之对应的表项，并将表项的索引存入 Index 寄存器

nop等待

把CP0_INDEX原有的值写入寄存器t1

判断如果t1的值小于0，跳转到NO_SUCH_ENTRY标签

如果t1的值不小于0，清空CP0_ENTRYHI和CP0_ENTRYLO0的值

nop等待

以 Index 寄存器中的值为索引，将此时CP0_ENTRYHI与CP0_ENTRYLO0的值写到索引指定的 TLB 表项中

将寄存器t1的值存回CP0_ENTRYHI中，返回

Thinking 2.6

x86架构中对内存的管理使用两种方式，即分段和分页。而在x86架构中内存被分为三种形式，分别是逻辑地址、线性地址和物理地址。通过分段可以将逻辑地址转换为线性地址，而通过分页可以将线性地址转换为物理地址

X86用到三个地址空间的概念：物理地址、线性地址和逻辑地址。而MIPS只有物理地址和虚拟地址两个概念。相对而言，段机制对大量应用程序分散地使用大内存的支持能力较弱。所以Intel公司又加入了页机制，每个页的大小是固定的（一般为4KB），也可完成对内存单元的安全保护，隔离，且可有效支持大量应用程序分散地使用大内存的情况。x86体系中，TLB表项更新能够由硬件自己主动发起，也能够有软件主动更新。

Thinking A.1

- PD_{base}=PT_{base}|PT_{base}>>9|PT_{base}>>18
- $PDE_{self\_mapping}$=PT_{base}|PT_{base}>>9|PT_{base}>>18|PT_{base}>>27

二、难点分析

Exercise 2.3 page_init函数的实现

根据题目，函数实现流程很清楚，难点主要在于对几个宏功能的正确理解和正确运用(Hint十分有用！)。

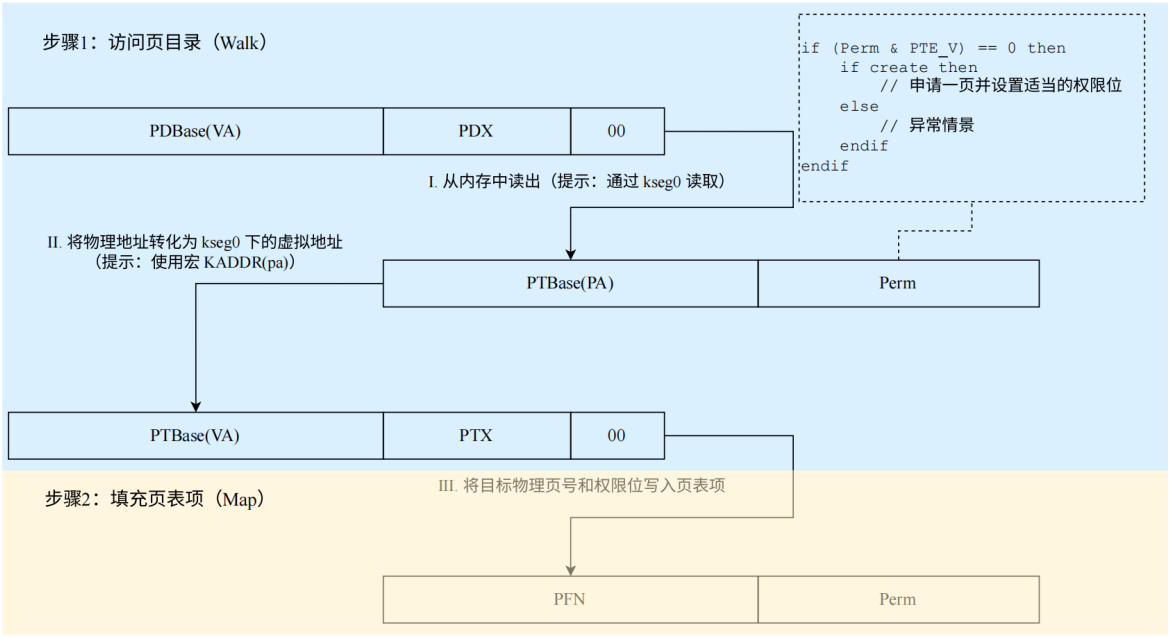
例如ROUND宏将freemem按照BY2PG对齐

PADDR宏将虚拟地址freemem映射为物理地址，进而才能找到小于 freemem 对应物理地址的物理内存

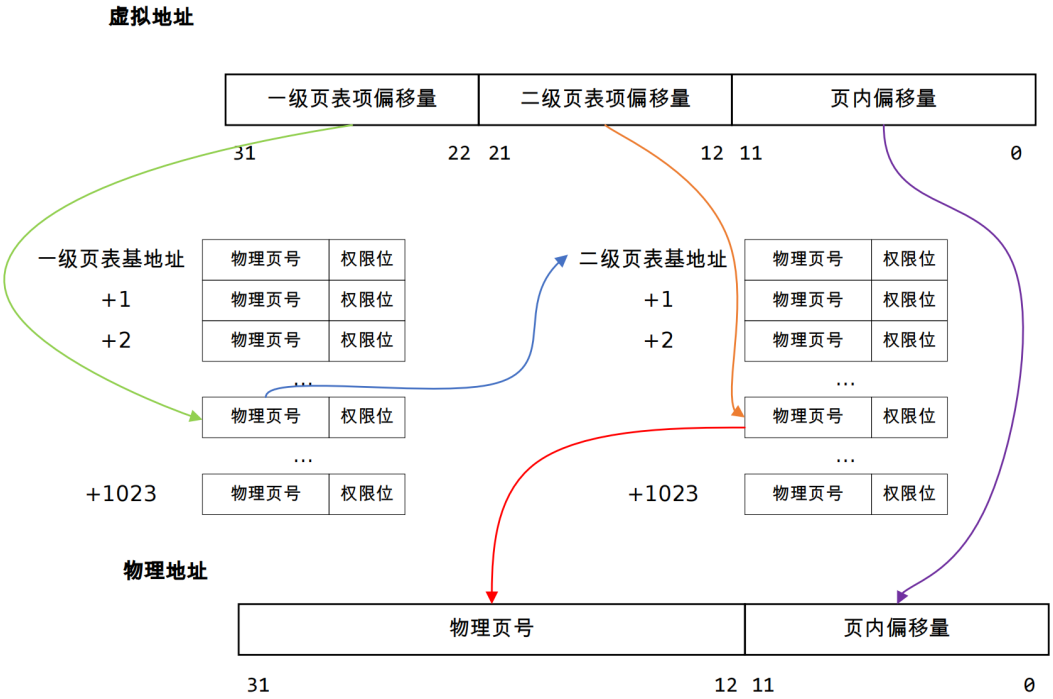
这些宏的定义和作用指导书中都有给出，需仔细阅读；另外还可在实际代码中找到这些宏的定义，便于更清楚的理解其是如何实现的

Exercise 2.6 pgdir_walk函数

难点在于，一是要根据指导书的图示清楚函数实现流程，二是对不同情况的分支判断



如果看这个图解，还不够明白流程，还可以先看看上文两级页表结构的地址变换机制的图解



此外还要注意指导书两处内容

- 设 pgdir 是一个 Pde * 类型的指针，表示一个一级页表的基地址，那么使用 pgdir+i 即可得到偏移量为 i 的一级页表项（页目录项）地址
- PDX(va) 可以获取虚拟地址 va 的 31-22位，PTX(va) 可以获取虚拟地址 va 的 21-12 位

根据以上信息，结合理论课上对二级页表的学习，实现该函数的地址变换的部分便很轻松了，剩下的便是查找到的二级页表不存在时的操作，前面Exercise 2.4编写的page_alloc函数以及page2pa函数便派上了用场，通过这两个函数创建二级页表，并让我们查找的页表项指向这个新创建的二级页表即可。

三、实验心得

这次实验使用了大量的宏去完成一些操作，刚开始做实验的时候，太过心急，没有仔细阅读指导书，导致对这些宏的作用都不熟悉，完成实验很困难。仔细阅读指导书，并根据实际代码理解这些定义的宏的功能后，一些"看不懂"的操作便恍然大悟，完成实验也轻松许多。

同时在这个过程中，也感受到了宏的魅力，无论是代码的可重用性，还是可读性，都有很好的提升，并且大大节约了代码量。

实验中对双向链表的操作，也让我对链表的操作更为熟悉。

此外，便是通过本次实验，我更清楚地明白了二级页表的机制，加深了理论课学习的理解。单单学习ppt的内容是不够的，只有实际去操作，才能更深入的掌握页式内存管理。