# lab2实验报告

## 思考题

### Thinking 2.1

都是虚拟地址

### Thinking 2.2

#### 宏实现链表的好处

提升了代码的可重用性

#### 三种链表性能差异

双向链表可以实现前插

单项链表无法实现，单项循环链表可以模拟实现前插，但是无法在常数时间完成，所以没有相关操作的宏定义。

如果给出前驱节点，三种链表都可以实现删除，但是如果不给出前驱节点，只有双向链表可以在常数时间实现删除操作。

对于可以在常数时间实现的操作，单项链表是最快的，因为只需要调整一个指针。

### Thinking 2.3

应该选择C。

pmap.h中给出了LIST_HEAD的定义：

```
LIST_HEAD(Page_list, Page);
```

这个宏的定义如下：

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

### Thinking 2.4

每个进程都有对应的地址空间，所以需要虚拟内存管理需要实现多个地址空间，需要将多个地址空间区分开来，所以ASID是必要的

ASID一共8位，所以最多可以有256片地址空间。

# Thinking 2.5

## tlb_invalidate和tlb_out的调用关系

tlb_invalidate调用tlb_out

## 请用一句话概括tlb_invalidate的作用

删除特定虚拟地址在TLB中的旧表项

## 逐行解释tlb_out中的代码

```
 3 LEAF(tlb_out)         声明一个叶子函数（无嵌套调用），名为tlb_out

 4 .set noreorder        禁止汇编器重排指令，确保严格顺序执行
 5         mfc0    t0, CP0_ENTRYHI      保存当前CP0_ENTRYHI的值到t0寄存器（备份）
 6         mtc0    a0, CP0_ENTRYHI      将参数a0（虚拟地址高位）写入CP0.ENTRYHI
 7         nop     确保mtc0完成
 8         /* Step 1: Use 'tlbp' to probe TLB entry */
 9         /* Exercise 2.8: Your code here. (1/2) */
10         tlbp    根据ENTRYHI的值在TLB中查找匹配条目，结果存入CP0.INDEX
11         nop     确保tlbp完成
12         /* Step 2: Fetch the probe result from CP0.Index */
13
14         mfc0    t1, CP0_INDEX    读取CP0到t1（如果找到条目，Index>=0,否则Index < 0）
15 .set reorder
16         bltz    t1, NO_SUCH_ENTRY        如果t1 < 0（未找到条目），跳转到NO_SUCH_ENTRY
17 .set noreorder
18         mtc0    zero, CP0_ENTRYHI        将EntryHI清零
19         mtc0    zero, CP0_ENTRYLO0       EntryLo0清零（物理页地位无效）
20         mtc0    zero, CP0_ENTRYLO1       将EntryLo1清零（物理页高位无效）
21         nop
22         /* Step 3: Use 'tlbwi' to write CP0.EntryHi/Lo into TLB at CP0.Index  */
23         /* Exercise 2.8: Your code here. (2/2) */
24         tlbwi           将清零后的ENTRYHI/ENTRYLO写入Index指定的TLB条目00
25 .set reorder    恢复汇编器指令重排
```

# Thinking 2.6

触发TLBMiss

pgdir_walk:根据页目录和虚地址寻找页表项，如果页表无效，重新分配新页表

分配新页面填写到页表项：pgdir_insert

取出相邻奇偶页填写到EntryLo，写回TLB：do_tlb_refill

## Thinking 2.7

| 特性 | MIPS | RISC-V |
| --- | --- | --- |
| **地址转换** | 混合分段+分页，固定内核映射 | 纯分页，全虚拟地址空间 |
| **TLB 管理** | 专用硬件指令（`tlbwr`、`tlbr`） | 软件管理（`sfence.vma` + CSR） |
| **异常处理** | 独立 TLB Refill 异常向量 | 统一异常入口，软件区分原因 |
| **权限控制** | 双模式（内核/用户） | 多级特权模式 + 细粒度页表权限 |
| **扩展性** | 固定设计，扩展困难 | 模块化分页方案，支持未来扩展 |

RISC-V 的设计更现代，强调软件灵活性和硬件中立性，而 MIPS 在传统嵌入式场景中依赖硬件优化。选择时需权衡性能需求与软件复杂度。

# 难点分析

### 物理内存管理

理解Page数组本身映射到所有可用物理内存，Page数组的结构体单元包含一个链表结构体和一个引用计数器。

包含链表结构体是为了便于构建空闲链表。

注意空闲链表的结构，链表结构体的两个指针，prev指向的是上一个结构体的next指针，next指向下一个结构体，注意这里不是对称的设计。而头结点是特殊的结构体，只包含一个指向第一个空闲Page的指针。

同时还要理解各种地址转换宏的含义

- page2pa，是把Page*指针转换程对应物理地址
- KADDR，是把物理地址转换成相应虚拟地址，这里的转换是当成kseg转换的，也就是直接+0x80000000，但是要先检查物理地址有没有超范围（物理空间小于虚拟地址空间）
- page2kva，嵌套使用上面两个，把Page*指针转换程成对应的虚拟地址

### 虚拟内存管理

其实可以随便用page_alloc分配出空闲的一页来当一级页表。

然后用pgdir_walk，来找二级页表，如果二级页表不存在，继续用page_alloc得到一页来当二级页表。然后pgdir_walk到这里就结束了，得到二级页表项的地址就完成使命了。

page_lookup就是需要找到va对应物理页控制块的地址，这里的va就是用户空间的va，同时将ppte指向对应二级页表项的地址，如果没有映射的话就返回NULL。

page_remove，删除va对物理地址的映射，如果存在这样的映射，那么对应物理页面的引用次数减少1。

- pgdir_walk
- page_insert
- page_lookup
- page_remove

# 实验体会

lab2最关键的就是理解各种含义的地址之间的转换。

首先要理解所有的内核代码都在kseg0里面。我们目前写代码就是为了使得用户空间可用，在用户空间可用之前，我们的所有代码和数据都是在kseg0段的，也就是说pages数组，这些页控制块本身的地址是在kseg0段的，Page*类型的指针范围是kseg0。但是每个Page映射的是一个物理页。从Page*到物理地址的转换就是page2pa。物理地址空间总共是64MB大小，这是所有可以用的空间。4GB地址空间就是要映射到这64MB大小的物理空间当中。如果知道某一页是kseg段的，那么将对应物理地址转换成虚拟地址，就需要用KADDR。

pages数组本身存放在0x80400000之后，这个地址是虚拟地址，使用page2pa宏可以将pages数据本身的虚拟地址转换成对应的物理地址。也就是说，在page2pa（page)到page2pa(page)+sizeof(struct Page)这一块物理地址上，存放着page[0]的pp_link数据和pp_res数据。而page[0]映射到物理地址是0x0~PAGESIZE，page[i]映射的物理空间是[i*PAGE_SIZE,i*PAGE_SIZE+PAGE_SIZE)，

i*PAGE_SIZE在实际代码中往往可以写成i<<PAGE_OFFSET，PAGE_OFFSET一般是12；