# 林子杰_20373980_lab2

## 一、思考题

### Thinking 2.1

我们编写的程序（如C、C++）等，指针变量使用的地址是虚拟地址。

我们编写的MIPS汇编程序，lw、sw等指令使用的是虚拟地址。

对于我们课程设计所使用的CPU，也是存在虚拟地址到物理地址的映射的。在课程网站上有kesg0、kseg1等虚拟地址段的映射方法，在此不赘述。

### Thinking 2.2

通过宏编写链表定义以及操作函数，不仅能够以库的形式在多个文件中实现复用，还可以看见，根据宏定义的特性，能够方便地根据类型定义链表，创建含有不同对象的链表，而不需要在每次想使用新的链表时进行重新定义。这都提高了代码的复用性。

至于实验环境中的"单向链表"与"循环链表"的实现，我在文件中好像并没有发现，于是自己上网查询了一下资料。首先，我发现我们使用的链表，用更加专业的术语来称呼应该是"双向无尾链表"。另外两种链表应该指的是"单向无尾链表"与"循环链表队列"。

- 单向无尾链表：单向无尾链表和我们平时接触的单向链表几乎一样，存在一个头节点，后面尾随着一个一个节点。单向无尾链表在每次插入删除时都需要遍历，适合作为堆栈使用
- 双向无尾链表：即我们使用的链表，在插入删除时，只需要了解其前驱或后驱节点即可插入删除，而不需要遍历。不过其ELF格式文件的大小、时空性能都会比单向无尾链表稍差。
- 循环链表队列：头尾相接的单向链表。其插入删除也需要进行查询，不过头尾相接的特性会有利于某些数据的存储和查询。其时空开销也比单向无尾链表大。

### Thinking 2.3

对照文件里定义的宏、在草稿纸上一步一步展开，其实不难得到结果，就是有些绕。展开如下：

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

## Thinking 2.4

`boot_map_segment()`函数在`mips_vm_init()`中被调用，后者的功能之一是建立二级页表映射机制，它通过下面这种方法调用了前者。

```
/* pgdir是一级页表的基地址，将[UPAGES,UPAGES+n)字节的虚拟地址
与[pages,pages+n)的物理地址建立映射关系，并且赋予权限PTE_R */
boot_map_segment(pgdir, UPAGES, n, PADDR(pages), PTE_R);
```

`boot_pgdir_walk()`函数在`boot_map_segment()`函数中被调用，前者可以返回虚拟地址对应的二级页表项的地址，后者通过修改这个地址的内容，即可达到映射的效果。

## Thinking 2.5

我们知道，TLB根据虚拟地址直接查找其中存储的页表，倘若有匹配，则不会再去多次访问内存了。但是这样可能会引发数据不安全的问题，因为可能会出现这样的情况：不同的进程，其相同的虚拟地址，映射到的物理页不同。显然，在这种情况下，一旦发生了进程切换且新的进程需要访问一个已存在于TLB的虚拟地址时，新的进程会得到错误的物理页面。为了解决这个问题，针对不同的进程，我们提供一个**ASID标识码**，来确保虚拟页查找时的进程专一性。

查询《IDT R30xx Family Software Reference Manual》后，得知ASID占据了EntryHi寄存器的6～11位共6位，故而R3000可容纳最大64个不同的地址空间。

## Thinking 2.6

查阅代码可知，`tlb_invalidate()`函数调用了`tlb_out()`,且进行了两次调用。两次调用的区别是是否将传入的地址进行了ASID码标记。

`tlb_invalidate()`的作用是：在保证对进程的跟踪下更新tlb。

下面对`tlb_out`中的汇编代码进行解释。

```
/* Exercise 2.10 */
LEAF(tlb_out)
//1: j 1b
nop
    /* 将CP0中EntryHi寄存器的内容写入k1 */
  mfc0  k1,CP0_ENTRYHI
    /* 将a0写入EntryHi寄存器，a0是一个虚拟地址 */
  mtc0  a0,CP0_ENTRYHI
    /* nop为延迟操指令 */
  nop
  /* 根据 EntryHi中的Key查找tlb中与之对应的表项，并将表项的索引存入Index寄存器；若未找到匹配项，则Index寄存器最高位被置1。 */
  tlbp
  nop
  nop
  nop
  nop
```

```
    /* 将Index寄存器的值写入k0寄存器 */
  mfc0  k0,CP0_INDEX
    /* 判断是否找到tlb表项。若未找到，k0最高位为1，小于零，直接跳转到NOFOUND标签 */
  bltz  k0,NOFOUND
  nop
    /* 找到，则将EntryHi和EntryLo寄存器都清空 */
  mtc0  zero,CP0_ENTRYHI
  mtc0  zero,CP0_ENTRYLO0
  nop
    /* 以Index寄存器中的索引为目标，将上面两个寄存器的内容写入该索引对应的表项 */
  tlbwi
NOFOUND:
  /* 不论如何，恢复现场 */
  mtc0  k1,CP0_ENTRYHI
  /* 跳转回去 */
  j ra
  nop
END(tlb_out)
```

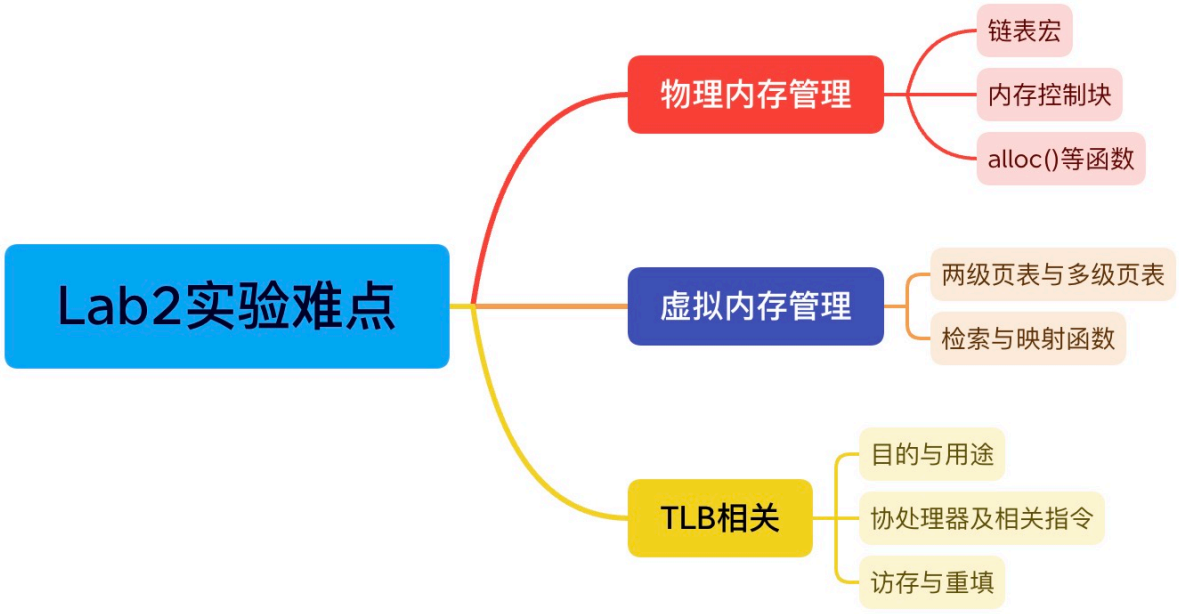
## Thinking 2.7

与二级页表相比，三级页表的页大小不变，但存储为变为64位，即8B，对于每一级页表的基地址，其偏移更多了（每一页所能存放的项也变少了）。首先计算三级页表的上一级，三级页表基地址为PTbase，则对应了PTbase>>12的页。想使得二级页表第一项（这里的项指的是64位一个的项）映射到三级页表第一页，二级页表的第一项应当相对三级页表第一项偏移PTbase>>12个项，即PTbase>>9个B（每一项8B），二级页表基地址为PTbase+PTbase>>9。一级页表相对于二级页表的计算同理，其基地址为PTbase+PTbase>>9+PTbase>>18。

## Thinking 2.8

x86采用了复杂的段式内存管理机制。进程仍旧是操作虚拟地址，通过分段的机制，映射线性地址中的段，从而从线性地址中取出页表索引，通过索引结合页表，取出物理地址。短式内存管理明显的优点是其内存分配更加弹性，能够更方便地分配内存空间，可以充分实现共享与保护，"能动性"是其相对于页式管理的优点（即mips的存储管理方式）。其缺点是容易产生内存碎片，造成浪费或是长期使用后难以装载进程。

# 二、实验难点




其实，具体到细节上，lab2的难点是很多的。既要把握核心内容，又要通读代码，保证不要顾此失彼。仅仅阅读指导书，以线性的方式学习知识，其实是不够的。代码中的一些宏、函数，可能都被"藏了起来"，需要我们自己去发掘观察，才能在看到某一行代码时，立刻反应过来它的作用。另外，关于lab2的核心内容，几个需要填写的函数的代码量还是很庞大的，倘若没有框架的提示，估计很难通过自己完成。

## 1.物理内存管理

MOS使用页式管理方法管理物理内存，具体使用链表来实现。此处的链表并不是我们上数据结构课时使用的简单链表，而是通过一系列的宏进行了封装。利用宏，MOS封装了一系列操作链表的方法，包括定义头节点、根据元素进行插入删除等、操作头尾元素等。通过填写代码，可以深刻体会到使用宏对链表操作进行封装的优点，也可以体会到C语言的一些灵活特性。

通过对最简单的链表进行扩充，我们得到了内存控制块，每个内存控制块即为一个Page结构体，npage个内存控制块的起始地址为`pages`，利用一个链表`page_free_list`记录空闲的物理内存。

我们还定义了一些列函数，如`page_init()`、`page_alloc()`等来管理这些内存控制块和链表，从而建立物理内存管理的体系。

## 2.虚拟内存管理

用户的进程启动时，并不是直接使用物理地址，而是通过虚拟地址来访问物理地址。如何将虚拟地址与物理地址建立联系？在MOS系统中，我们使用两级页表结构来构建这一关系。首先我们需要明白，一个虚拟地址是如何访问物理地址从而得到相应数据的。我们要清楚虚拟地址每一段的意义与使用方法，也要理解每一级页表中所存储数据不同段的意义与使用方法；同时，我们也要掌握数据单位的转换，以便更好的在概念之间进行切换。

### 3.TLB相关

TLB虽然和地址映射有比较大的关系，但是作为一个重要的硬件，还是将它单独作为一个主要的知识点进行理解。TLB更接近底层，更接近协处理器、中断，了解它如何进行存取，如何与协处理器进行交互、使用什么样的mips指令，有助于我们后面学习中断的相关知识。

## 三、体会与感想

lab2的难度再次提示。直到写完这次报告，我感觉我对内存管理的认识还处于一个初始的、零碎的阶段，并没有形成完整的框架。这一部分是硬件还是软件的工作？这里为什么要这样映射？恐怕很多问题我还没能完全弄明白。而对于代码中的部分函数，我也仅仅是停留在"能够读懂每一行在干什么"的阶段，一旦上升到整个函数的层面，我就很难操作系统中找到它的位置了。对于这个问题，我想我可能还需要阅读更多的书籍，来建立一个完整的知识结构。

lab2引证了指导书lab1中的一句话，"这一部分知识点过于零散.....没有很好的体系....."。的确，这一部分的学习确实让人感到吃力。对于不明白的地方，还是应该要和同学及时交流，不能闭门造车，这样反倒会浪费很多时间。

另外，将本次试验的笔记附在下方。

# OS Lab2 内存管理

## 一、虚拟地址映射

理论课程解释过为什么现代CPU一般都是操作虚拟地址。在实验课中我们将对此进行体会思考。

### Thinking 2.1

我们编写的程序（如C、C++）等，指针变量使用的地址是虚拟地址。

我们编写的MIPS汇编程序，lw、sw等指令使用的是虚拟地址。

对于我们课程设计所使用的CPU，也是存在虚拟地址到物理地址的映射的。在课程网站上有kesg0、kseg1等虚拟地址段的映射方法，在此不赘述。

## 二、启动

我们需要在启动的时候对地址映射做一些初始化的工作。这些初始化的工作在我们之前接触过的main函数中；main函数调用了`mips_init()`函数进行初始化工作。它调用了`mips_detect_memory()`、`mips_vm_init()`等共五个函数来完成初始化。

- `mips_detect_memory()`:本函数的作用为对一些和内存相关的变量进行初始化，以便之后使用。
- `mips_vm_init()`:按照注解，这个函数用于设置两级页表。其中第一步，使用同样定义在本文件中的`alloc()`函数建立一级页表（alloc的作用是分配空间并返回其虚拟地址的初始位置）。完成这一步之后，`boot_pgdir`变量存放的就是对应虚拟地址的初始地址。

  然后，为管理物理内存用到的`page`结构体按页分配物理内存；在之后，为管理进程用到的`Env`结构体按页分配物理内存。
- `page_init`:用于初始化Page结构体并且初始化空闲链表。

- 另外两个函数用于检测我们填写代码的正确性。

我们在此完成内存管理所需要启动阶段做的初始化工作。

## 启动的时候是如何创建二级页表的？（补充于2022.4.8）

在之前填写代码的过程中，发现自己是什么都看不明白；又由于时间紧迫，故而先照着已有代码抄了一遍。最近有一些空闲时间，变回来尝试着仔细阅读了一下代码，并且在此补充一下我目前对于系统启动时创建二级页表的方式的理解。

首先，初始化的程序位于`main.c`，但这个C文件仅仅是调用了`init.c`中的函数`mips_init()`来实现初始化；观察`mips_init`，发现这个函数也没有做太多实质性的工作，其之所以能够实现初始化还是因为调用了`mips_detect_memory()`和`mips_vm_init()`函数。前者用于根据系统内存大小、页大小来初始化物理地址、物理页数和扩展内存，是一个比较简单的函数；后者的代码才是重头戏，是创建二级页表的"元勋"。

`mips_vm_init()`里面有很多变量，同时也调用了很多函数。我们逐一理解其每一步的含义。在这里，我会省去一些我认为对于理解不起关键作用的代码。

第一个比较关键的步骤是：

```
Pde *pgdir
//alloc(n, align, clear)，地址对其align后申请一片n字节大小的空间，倘若clear有效则将这一片空间清空
pgdir = alloc(BY2PG, BY2PG, 1);
boot_pgdir = pgdir
```

第一句声明了一个`Pde`类型的指针名为`pgdir`。教程中有介绍，`Pde`用于表示一级页表，`pgdir`就是这个页表项内存位置的指针。不过，很显然这个指针目前没有指向任何东西。为了让它能够发挥作用，我们使用`alloc()`函数请求了一片空间，并且返回这一片空间的初始，令`pgdir`等于这一初始地址。`alloc()`函数的用途已经写在注释当中。这样一来，我们就成功地在内存中为一级页表申请了一片空间，接下来就可以着手在里面填写内容了。

看到这里，你可能会问：这个`alloc()`函数是怎么做到分配内存空间的呢？为了保证阅读的流畅，我们现在还是关注当前这个函数。

在做第二步之前，我们先回想一下我们管理页用到数据结构：对于每个页，我们创建一个`Page`结构体进行管理。这些`Page`存放在物理内存中，用一个`pages`数组进行管理。为了进行拓展，我们还需要将`pages`虚拟地址对应的物理地址和虚拟地址UPAGES进行映射。

```
pages = (struct Page *)alloc(npage * sizeof(struct Page), BY2PG, 1);
n = ROUND(npage * sizeof(struct Page), BY2PG);
/*
boot_map_segment(Pde *pgdir, u_long va, u_long size, u_long pa, int perm)
这个函数的作用的是将[va,va+size)的虚拟地址映射到[pa,pa+size)的物理地址上
pgdir是一级页表内存的指针，va是虚拟地址起始位置，size是虚拟空间的大小，pa是物理地址起始位置，perm为权限位
*/
boot_map_segment(pgdir, UPAGES, n, PADDR(pages), PTE_R);
```

可以看到，我们同样为数组`pages`开辟了一段空间，用于存放其管辖的诸多结构体。不过，这次开辟空间的大小是根据`Page`结构体的数量来决定的。然后，我们可以看到一个`n`和一个函数`boot_map_segment()`。不过，只写出其功能，其内容在此处还是按下不表。

下面还有一段为全局数组`envs`申请内存空间并进行映射的代码，不过这一个数组是用于进程管理的，而且和上面的代码段几乎是一模一样，~~就是换了个数组名字而已~~。就不多赘述了。

# 三、物理内存管理

## 1.链表宏

在介绍这一部分之前，首先介绍一下一种宏定义的写法：

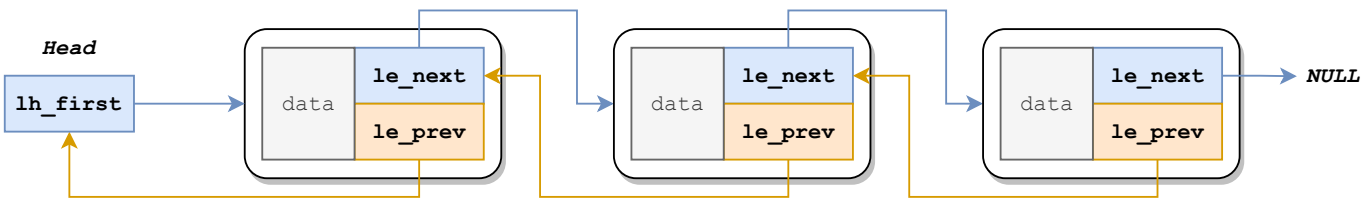
```
do {
  ......
} while(0)
```

这是一种专门针对代码块的"安全"、"约定俗成"的写法。为了使我们用宏来插入代码段等宏字段不产生安全问题，人们想出了这种优秀的写法，能够规避一些语法和逻辑上的问题。在这里先做记录，以免之后再次看到时不知所云。

在这一节中，我们需要阅读一个名为`queue.h`的文件，里面定义了一系列有关与双向链表操作方法的宏。其操作的基本单元是一个链表头部宏和一个结构体宏。

```
//链表头部宏
#define LIST_HEAD(name, type)    \
struct name {                    \
    struct type *lh_first;       \
}
//结构体宏
#define LIST_ENTRY(type)         \
struct {                         \
    struct type *le_next;        \
    struct type **le_prev;       \
}
```

头节点比较特殊，和其后面连接的节点不是同一类结构体。`*lh_first`存储了第一个子结点的地址；`*le_next`储存了当前结点的下一个结点的地址，`**le_prec`存储了当前结点的前一个结点的`*le_next`的地址。这种组织方式比较特殊，用图片表示是这样的。



在此基础上，我们会实现取头节点、在某一结点前后插入结点、在头尾插入结点等一系列函数宏。其中一些已经实现，另外`LIST_INSERT_AFTER`和`LIST_INSERT_TAIL`的宏需要我们自己实现。

## 2.内存控制块

在MOS中，我们通过**npage**个**内存控制块**维护内存，内存控制块即是**Page结构体**。npage个物理页面和Page结构体是一一顺序对应的（索引从0开始）。

在初始状态下，将所有结构体都插入一个链表中，此链表被称为空闲链表。每当有进程需要被分配内存时，就会将链表**头部的内存控制块对应的物理内存**分配出去，同时将该内存控制块从链表头部剔除；同样地，当物理页使用完毕时，其内存控制块又会被重新插到链表的头部。

## 3.相关函数

此处，用于物理内存管理的函数有很多个。他们主要是围绕着一个"管理空闲页的链表"展开。

- `page_init()`:初始化内存管理链表，这个函数在`mips_vm_init()`执行后才被调用——也就是说，`freemem`等值已经有了初始值了。有一个名为`&page_free_list`的指针作为链表的头节点用于管理空闲页面。

```
void page_init(void)
{
    //利用宏定义初始化page_free_list，将其指向NULL
    LIST_INIT(&page_free_list);
    //将freemem按照页进行对齐
    ROUND(freemem, BY2PG);
    /* 还记得freemem的意思吗？阅读过mips_vm_init()函数后你就会发现，freemem表示
    [0x80400000,freemem)的虚拟地址已经被使用了，这段虚拟地址也对应了一些物理页，这
    些页面在启动的过程中已经被用于存放二级页表了，故而其相当于已经被使用了，我们要把这
    些物理页面标记为已经使用。pp_ref置1表示本页已经被使用，现在要做的就是把freemem以
    下的虚拟地址对应的物理页标记为使用过 */
    struct Page *cur = pages;
    /* "page to kernel virtul address"，cur指向pages数组的虚拟地址，用page2va()
    转换为u_long类型，用于遍历。若cur的地址在freemem之下，将其指向的Page结构体的pp_ref
    属性全部置1，标记为使用过 */
    for(;page2kva(cur) < freemem;cur++)
    {
        cur->pp_ref = 1;
    }

    /* 剩下的页面就是空闲的页面了，首先将cur和freemem指向的物理页对齐，并赋给他这一页的地址
    接下来cur不断增长，在其增长至地址对应的物理页数等于最大物理页编号之前，将每个增长时经过的
    物理页标记为没有使用过，并插入page_free_list身后的链表，用于管理 */
    for (cur = &pages[PPN(PADDR(freemem))];page2ppn(cur) < npage;cur++)
    {
        cur->pp_ref = 0;
        LIST_INSERT_HEAD(&page_free_list, cur, pp_link);
    }
}
```

## Thinking 2.2

通过宏编写链表定义以及操作函数，不仅能够以库的形式在多个文件中实现复用，还可以看见，根据宏定义的特性，能够方便地根据类型定义链表，创建含有不同对象的链表，而不需要在每次想使用新的链表时进行重新定义。这都提高了代码的复用性。

至于实验环境中的"单向链表"与"循环链表"的实现，我在文件中好像并没有发现，于是自己上网查询了一下资料。首先，我发现我们使用的链表，用更加专业的术语来称呼应该是"双向无尾链表"。另外两种链表应该指的是"单向无尾链表"与"循环链表队列"。

- 单向无尾链表：单向无尾链表和我们平时接触的单向链表几乎一样，存在一个头节点，后面尾随着一个一个节点。单向无尾链表在每次插入删除时都需要遍历，适合作为堆栈使用
- 双向无尾链表：即我们使用的链表，在插入删除时，只需要了解其前驱或后驱节点即可插入删除，而不需要遍历。不过其ELF格式文件的大小、时空性能都会比单向无尾链表稍差。
- 循环链表队列：头尾相接的单向链表。其插入删除也需要进行查询，不过头尾相接的特性会有利于某些数据的存储和查询。其时空开销也比单向无尾链表大。

## Thinking 2.3

对照文件里定义的宏、在草稿纸上一步一步展开，其实不难得到结果，就是有些绕。展开如下：

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

# 四、虚拟内存管理

## 1.两级页表结构

MOS使用两级页表对`kuseg`虚拟地址进行管理。在一级页表中，我们的虚拟内存地址分为"页表偏移"和"页内偏移"。在二级页表结构中，其实是将一级页表进行了进一步的切分。MOS中的虚拟地址有32位，一级页表占高10位，二级页表占中10位，页内偏移占低12位。

在`include/mmu.h`中存放了两个宏用于获得偏移量：`PDX(va)`可以获得虚拟地址`va`的31-22位；而`PTX(va)`可以获得虚拟地址的21-12位。

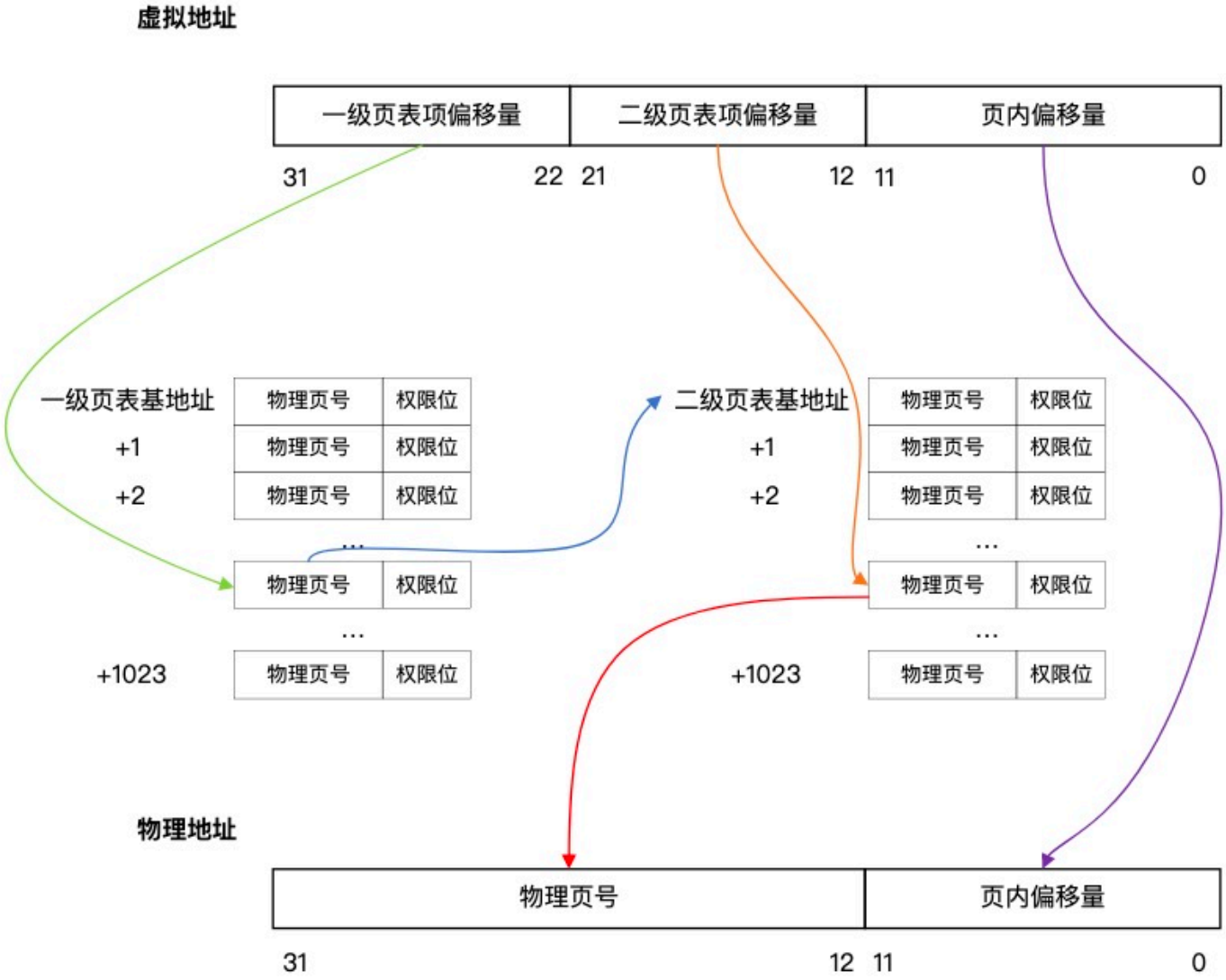
在`mmu.h`中定义了`Pde`表示一个一级的页表项，用`Pte`表示一个二级的页表项。倘若一个`Pde`类型的指针名为`pgdir`那么用`pgdir + i`就可以获得偏移量为`i`的页表项地址。

## 2.启动相关函数

此处有两个函数：`boot_map_segment()`和`boot_pgdir_walk()`。这两个函数是在内核启动过程中，用于初始化两级页表的。

- `boot_map_segment()`:将虚拟地址区间 [va, va + size − 1] 映射到物理地址区间 [pa, pa + size − 1]。这个函数调用了`boot_pgdir_walk()`。
- `boot_pgdir_walk()`:此函数返回一级页表基地址 pgdir对应的两级页表结构中va这个虚拟地址所在的二级页表项。如果 create 不为 0 且对应的二级页表不存在，则会使用 alloc 函数分配一页物理内存用于存放。

这一部分的代码量突然增大，需要填空的函数也变得很多，尽管指导书上给出了很多提示性的内容甚至是框架，我还是认为有必要尝试写一下源码的分析过程。下面，我将尽可能借助这一映射图来解释代码。




```
/* 传入的第一个参数是某一级页表项的基地址，va是一个虚拟地址。此函数的作用是返回一个二级页表项即"Pte
*"。倘若该页表项不存在则为其申请一个空间并创建 */
static Pte *boot_pgdir_walk(Pde *pgdir, u_long va, int create)
{
    Pde *pgdir_entryp;
    Pte *pgtable, *pgtable_entry;
    /* PDX(va)用于获得虚拟地址va的31-22位，pgdir_entryp为一级页表项的虚拟地址
    此处，PDX(va)相当于取出了虚拟地址的一级地址偏移，加上pgdir这一基地址，就能得到
```

```
    va所对应的二级页表项的位置 */
    pgdir_entryp = pgdir + PDX(va);
    //判断二级页表是否存在
    if ((*pgdir_entry & PTE_V) == 0) {
        //判断是否需要创建二级页表
        if (create) {
            /* 开辟内存空间，alloc返回虚拟地址，将其转成物理地址赋给va在一级页表中
            所对应的页表项。注意，*在此处是取值的意思，即修改pgdir中某32位的内容 */
            *pgdir_entryp = PADDR(alloc(BY2PG, BY2PG, 1));
            //设置权限
            *pgdir_entryp = (*pgdir_entryp) | PTE_V | PTE_R;
        } else return 0; // exception
    }
    /* PTX(va)用于获得虚拟地址va的21-12位。返回该二级页表入口的虚地址。将va对应的
    二级页表项的地址加上二级页表偏移，即可得到物理页号的地址 */
    return ((Pte *)(KADDR(PTE_ADDR(*pgdir_entryp)))) + PTX(va);
}
```

```
/* 传入的第一个参数是某一级页表项的基地址，va是一个虚拟地址，size是这个页表项对应的虚拟地址的空间大小，
pa是物理地址。此函数用于映射物理地址和虚拟地址 */
void boot_map_segment(Pde *pgdir, u_long va, u_long size, u_long pa, int perm)
{
    int i, va_temp;
    Pte *pgtable_entry;
    //循环的开始，先将size和页大小进行对齐
    for (i = 0, size = ROUND(size, BY2PG); i < size; i += BY2PG) {
        //向boot_pgdir_walk()函数传入页表基地址、虚拟地址和create，返回二级页表虚地址
        pgtable_entry = boot_pgdir_walk(pgdir, va + i, 1);
        //映射，虽然我也不知道映射的原理
        /* 2022.4.16补充，我今天知道映射的原理了。由于boot_pgdir_walk()函数返回的是二
        级页表项已经偏移过的内容，即这一地址所存放的东西，就是物理页号和权限位，直接对这个地址
        的内容进行修改，就能够完成映射 */
        *pgtable_entry = (pa + i) | perm | PTE_V;
    }
}
```

这里出现了很多"乱七八糟"的函数，应该要多多回顾，多多阅读代码。

## 3.进程运行相关函数

在这一个部分，我们需要填写`pgdir_walk()`函数和`page_insert()`函数。这两个函数和上面填写过的函数很相似，只是他们不是在启动时被调用，而是会在普通进程运行时被调用。

```
/* 此函数的作用也是根据虚拟地址va，将**ppte修改为该虚拟地址对应的二级页表项的地址，不过这个函数
是在进程运行中时使用的，寻址允许失败 */
int pgdir_walk(Pde *pgdir, u_long va, int create, Pte **ppte)
{
    //得到一级页表偏移后地址，这个地址存放的内容是某个二级页表的基地址
```

```
    Pde *pgdir_entry = pgdir + PDX(va);
    struct Page *page;
    int ret;

    //检查该二级页表是否存在，不存在则创建，使用page_alloc()函数开辟空间
    if ((*pgdir_entry & PTE_V) == 0) {
        if (create) {
            //page变量用于开辟物理页，失败则返回错误码
            if ((ret = page_alloc(&page)) < 0) return ret;
            //成功，此时page已经申请到一个空间，将其地址赋予权限，填入此一级页表项
            *pgdir_entry = (page2pa(page)) | PTE_V | PTE_R;
          page->pp_ref++;
    } else {
            *ppte = 0;
            return 0;
        }
    }
    //同上，返回某一二级页表偏移后的地址，这一地址的内容就是物理页号和权限位
    *ppte = ((Pte *)(KADDR(PTE_ADDR(*pgdir_entry)))) + PTX(va);
    return 0;
}
```

```
/* 此函数的作用是，将va这一虚拟地址映射到内存中pp对应的物理页面，并且赋予权限 */
int page_insert(Pde *pgdir, struct Page *pp, u_long va, u_int perm)
{
    u_int PERM;
    Pte *pgtable_entry;
    PERM = perm | PTE_V;
    /* 使用pgdir_walk()函数，根据va修改pgtable_entry的值，使之指向va对应的二级页表项
    的地址，这里将create置为0，是为了检查而非创建 */
    pgdir_walk(pgdir, va, 0, &pgtable_entry);
    if (pgtable_entry != 0 && (*pgtable_entry & PTE_V) != 0) {
    /* 检查va是否已经和另一物理页面存在映射关系，即其对应的二级页表项的内容转换为物理页后
    是否和我们要映射的目标物理页的地址相同，如果已经存在映射关系，则使用page_remove()将其
    解除 */
    if (pa2page(*pgtable_entry) != pp) {
      page_remove(pgdir, va);
    } else  {
            tlb_invalidate(pgdir, va);
            *pgtable_entry = (page2pa(pp) | PERM);
            return 0;
        }
    }
    tlb_invalidate(pgdir, va);
    /* 再次指向该虚拟地址va所对应的二级页表项，倘若其中未存在映射，则分配页，并进行映射，
    映射失败，则返回错误码 */
    if(pgdir_walk(pgdir, va, 1, &pgtable_entry) != 0)
    {
        return -E_NO_MEM;
```

```
    }
    //赋予权限
    *pgtable_entry = page2pa(pp) | PERM;
    //标记页面非空闲
    pp->pp_ref++;
    return 0;
}
```

## Thinking 2.4

`boot_map_segment()`函数在`mips_vm_init()`中被调用，后者的功能之一是建立二级页表映射机制，它通过下面这种方法调用了前者。

```
/* pgdir是一级页表的基地址，将[UPAGES,UPAGES+n)字节的虚拟地址
与[pages,pages+n)的物理地址建立映射关系，并且赋予权限PTE_R */
boot_map_segment(pgdir, UPAGES, n, PADDR(pages), PTE_R);
```

`boot_pgdir_walk()`函数在`boot_map_segment()`函数中被调用，前者可以返回虚拟地址对应的二级页表项的地址，后者通过修改这个地址的内容，即可达到映射的效果。

# 五、访问与TLB重填

## 0.为什么需要TLB?

我们知道，TLB是类似于Cache的高速缓冲存储器，它有助于加快我们的页面切换速度。但是，处于什么原因，导致我们需要在CPU和内存之间增加一级TLB呢？我认为有必要提前介绍一下。

众所周知，我们在编程的时候使用的是虚拟地址（逻辑地址），CPU不能根据这个地址直接取值，所以CPU首先要将虚拟地址转化为物理地址，再到内存中取值。没有页表的时候，CPU直接通过硬件转化访问内存即可，这需要我们访问一次内存；倘若我们拥有一级页表，那么CPU的虚拟地址就不能直接找到物理地址，首先CPU要访问内存中存好的一级页表，根据虚拟地址的页表项偏移，取出物理页号和虚拟地址中的页内偏移进行拼接，得到物理地址，再根据这个物理地址访问内存。可见，一级页表的存在虽然便于我们管理进程，但是它引起了内存访问次数++的开销——二级页表相比于一级页表，还要多访问一次内存（需要查询内存中的二级页表）。这样，CPU来回访问内存带来的时间开销可能会成为无法忍受的代价（因为CPU访问内存本来就很耗时），故而在其中增加一个访问速度很快的中间级——TLB，来解决这一矛盾。

当然，TLB作为高速缓冲存储器，其价格比较高，使用较大的TLB的物质成本也会比较大，故而我们常常会根据局部性原理设计算法，保证TLB中所存储的页面是经常被访问的，以此提高TLB的效率。

## 1.TLB前置知识

**内存管理相关的CP0寄存器：**

| 寄存器序号 | 寄存器名 | 用途 |
| --- | --- | --- |
| 8 | BadVaddr | 保存引发地址异常的虚拟地址 |
| 10、2 | EntryHi、EntryLo | 所有读写 TLB 的操作都要通过这两个寄存器 |
| 0 | Index | TLB 读写相关需要用到该寄存器 |
| 1 | Random | 随机填写 TLB 表项时需要用到该寄存器 |

**TLB硬件结构：**

TLB由很多表项组成。每一个TLB表项都有64位，其中高32位是key，低32位是Data。

**EntryHi和EntryLo：**

对应到TLB的Key与Data的、存在于CP0中的寄存器。很显然，我们在使用快表访问内存的的时候，可能会遇到缺失的情况。这两个存在于CP0中的寄存器为我们提供了缺失中断时处理中断的可能。

EntryHi寄存器对应的内容是Key，Key又分为VPN（虚拟页号）和ASID（地址空间标识符）。

EntryLo寄存器对应的内容是PFN（物理编帧号）、N（是否经过Cache访存）、D（咳血脏位）、V（地址有效位）、G（全局位）。

Key和Value就是相互映射的键值对。

# 2.MIPS中TLB相关的指令

介绍一些MIPS中有关TLB操作的指令。和上学期的计组课程中的大部分指令一样，这些有关TLB的指令也是通过操作并读写寄存器来发挥作用的。下面提到的一些寄存器，可以在上面的表格中查找他们的作用。

- **tlbr**：以 Index 寄存器中的值为索引,读出 TLB 中对应的表项到 EntryHi 与 EntryLo。TLB表项是64位的，其高低32位的规范分别对应了EntryHi和EntryLo两个寄存器的规范。
- **tlbwi**：以 Index 寄存器中的值为索引,将此时 EntryHi 与 EntryLo 的值写到索引指定的 TLB 表项中。
- **tlbwr**：将 EntryHi 与 EntryLo 的数据随机写到一个 TLB 表项中（此处使用 Random 寄存器来“随机”指定表项，Random 寄存器本质上是一个不停运行的循环计数器）。
- **tlbp**：根据 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（若未找到匹配项，则 Index 最高位被置 1）。

# Thinking 2.5

我们知道，TLB根据虚拟地址直接查找其中存储的页表，倘若有匹配，则不会再去多次访问内存了。但是这样可能会引发数据不安全的问题，因为可能会出现这样的情况：不同的进程，其相同的虚拟地址，映射到的物理页不同。显然，在这种情况下，一旦发生了进程切换且新的进程需要访问一个已存在于TLB的虚拟地址时，新的进程会得到错误的物理页面。为了解决这个问题，针对不同的进程，我们提供一个**ASID标识码**，来确保虚拟页查找时的进程专一性。

查询《IDT R30xx Family Software Reference Manual》后，得知ASID占据了EntryHi寄存器的6～11位共6位，故而R3000可容纳最大64个不同的地址空间。

## Thinking 2.6

查阅代码可知，`tlb_invalidate()`函数调用了`tlb_out()`,且进行了两次调用。两次调用的区别是是否将传入的地址进行了ASID码标记。

`tlb_invalidate()`的作用是：在保证对进程的跟踪下更新tlb。

下面对`tlb_out`中的汇编代码进行解释。

```
/* Exercise 2.10 */
LEAF(tlb_out)
//1: j 1b
nop
    /* 将CP0中EntryHi寄存器的内容写入k1 */
  mfc0  k1,CP0_ENTRYHI
    /* 将a0写入EntryHi寄存器，a0是一个虚拟地址 */
  mtc0  a0,CP0_ENTRYHI
    /* nop为延迟操指令 */
  nop
  /* 根据 EntryHi中的Key查找tlb中与之对应的表项，并将表项的索引存入Index寄存器；若未找到匹配项，则Index寄存器最高位被置1。 */
  tlbp
  nop
  nop
  nop
  nop
    /* 将Index寄存器的值写入k0寄存器 */
  mfc0  k0,CP0_INDEX
    /* 判断是否找到tlb表项。若未找到，k0最高位为1，小于零，直接跳转到NOFOUND标签 */
  bltz  k0,NOFOUND
  nop
    /* 找到，则将EntryHi和EntryLo寄存器都清空 */
  mtc0  zero,CP0_ENTRYHI
  mtc0  zero,CP0_ENTRYLO0
  nop
    /* 以Index寄存器中的索引为目标，将上面两个寄存器的内容写入该索引对应的表项 */
  tlbwi
NOFOUND:
  /* 不论如何，恢复现场 */
  mtc0  k1,CP0_ENTRYHI
  /* 跳转回去 */
  j ra
  nop
END(tlb_out)
```

# 六、多级页表与页目录自映射

倘若有一个进程使用了4GB的空间（这恰好是我们的MOS能够给一个进程分配的最大空间的大小），那么我们需要将一级页表（页目录）和二级页表（页表项）存入内存。根据一般的思维，我们可以分配一段连续空间来存放一级页表，然后再分配一段连续空间存放二级页表，这种情况下，我们需要4MB（二级页表项占用的空间，1024*4KB）+4KB（一级页表项占用的空间，1024*4B）。这样好像有点丑陋，我们能不能有一个办法，将我们的一级页表放在一个特殊的位置，使得这个一级页表的内容恰好和某个二级页表表项的内容相同呢?

答案是：可行。

在介绍方法之前，首先我们了解一下二级页表的存取方式。要当我们拿到一个虚拟地址时，我们首先根据这个虚拟地址的最高十位，也就是一级页表项偏移，在一级页表项中查找对应的32位（一共有1024个32位，也就是4KB空间，刚好是一页的大小）。这32位中有一段非常关键，即物理页号。物理页号可以帮我们查询二级页表所在的基地址，二级页表中每一个表项又是4KB大小，故而我们再从虚拟地址中取中间十位，作为偏移。这样，又可以取出一个32位的数据，而这32位数据的高20位，就是物理页号（为什么是20位？因为我们的物理地址有32位，且物理地址是页式存储，每页4KB，故而一共有1M页，即2^20页，对应需要20位进行寻址）。将20位物理页号和虚拟地址中的页内偏移拼接起来，就得到了物理地址。

我们再来看看页目录自映射。自映射条件下，首先，我们希望通过一个物理地址可以找到页目录；然后我们希望找到的页目录的第一项（32位）存放的物理地址可以找到**页表项**的物理页号。我们知道，二级页表的大小为4MB（1024*1024*32B），共有1024*1024个项，对应了1024*1024页，且这一部分的映射是线性的。那么就不难理解**页表项**的每一项存的是什么：一个32位的包含物理页号的数据。页表项存储的32位的内容，尤其是物理页号是根据什么来决定的？答案是，其映射到的是，自身地址右移12位的物理页。这个时候，我们突然想起来，既然4MB大小的空间能映射到所有页上，那么这4MB中必定有一项存储的32位，可以映射到页表项的第一页（即第一个二级页表）；必定有连续的1024个32位（即一页），将页表项的每一页都映射到，这就是我们要寻找的页目录的基地址了。如何寻找这一地址？假设1024个页表项的第一项对应的地址为PTbase，那么它映射到的物理页为第一页；那么PTbase对应的物理页则是第PTbase>>12页，即相对于第一页偏移了PTbase>>12页。那么显然，映射到PTbase对应的页的那32位，也要相对于第一个32位偏移PTbase>>12个32位，即PTbase>>10个B。故而最终得到的页目录地址为PTbase+PTbase>>10。

## Thinking 2.7

与二级页表相比，三级页表的页大小不变，但存储为变为64位，即8B，对于每一级页表的基地址，其偏移更多了（每一页所能存放的项也变少了）。首先计算三级页表的上一级，三级页表基地址为PTbase，则对应了PTbase>>12的页。想使得二级页表第一项（这里的项指的是64位一个的项）映射到三级页表第一页，二级页表的第一项应当相对三级页表第一项偏移PTbase>>12个项，即PTbase>>9个B（每一项8B），二级页表基地址为PTbase+PTbase>>9。一级页表相对于二级页表的计算同理，其基地址为PTbase+PTbase>>9+PTbase>>18。

## Thinking 2.8

x86采用了复杂的段式内存管理机制。进程仍旧是操作虚拟地址，通过分段的机制，映射线性地址中的段，从而从线性地址中取出页表索引，通过索引结合页表，取出物理地址。短式内存管理明显的优点是其内存分配更加弹性，能够更方便地分配内存空间，可以充分实现共享与保护，"能动性"是其相对于页式管理的优点（即mips的存储管理方式）。其缺点是容易产生内存碎片，造成浪费或是长期使用后难以装载进程。