操作系统第三次实验报告-Lab2

以下为学生闫思桥(19241091)的$Lab2$实验报告

$Part1$ 实验思考题

Thinking 2.1

请你根据上述说明，回答问题：

- **在我们编写的程序中，指针变量中存储的地址是虚拟地址还是物理地址？**
- **MIPS 汇编程序中lw, sw使用的是虚拟地址还是物理地址？**

MyAnswer

在我们编写的程序中，指针变量中存储的地址是虚拟地址。可以这么理解，程序代表用户希望实现的逻辑关系，但是实际物理存储中并不会真的按照逻辑中的位置去存储，逻辑中的地址连续的，物理中并不一定连续。

MIPS汇编程序中，lw，sw使用的是虚拟地址。因为汇编和C最终都是机器码，都是在CPU上跑的程序，CPU只发出虚拟地址，所以他们编程时都是虚拟地址。

附上指导书里一张很关键的图。（lab3也可以回头看）

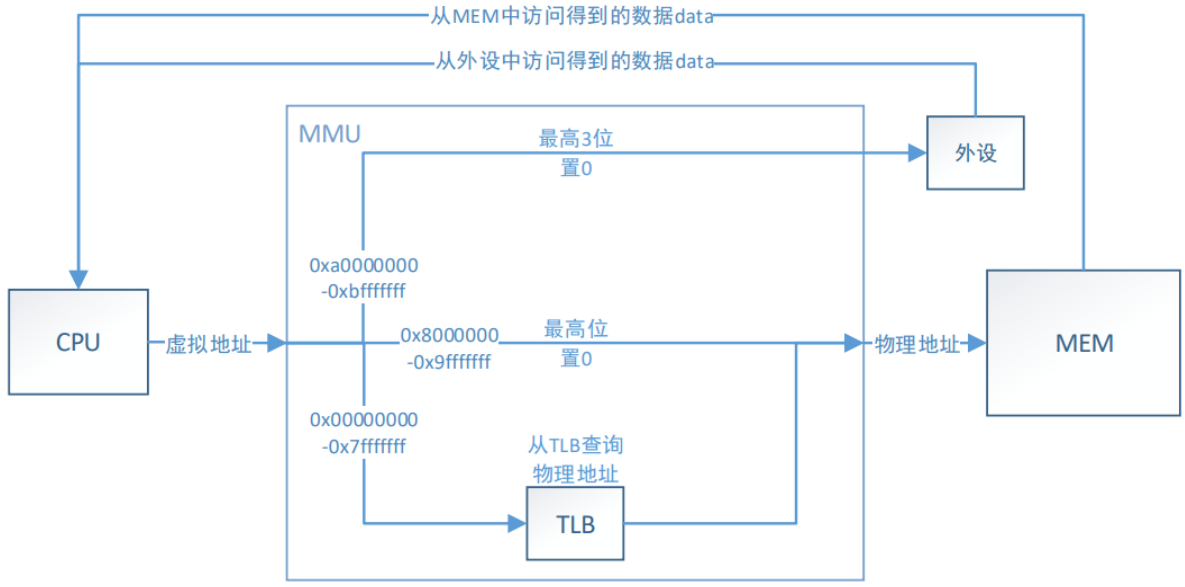


图 2.1: cpu-tlb-memory 关系

Thinking 2.2

- **请从可重用性的角度，阐述用宏来实现链表的好处。**
- **请你查看实验环境中的 /usr/include/sys/queue.h，了解其中单向链表与循环链表的实现，比较它们与本实验中使用的双向链表，分析三者在插入与删除操作上的性能差异。**

MyAnswer

- 用宏来实现链表的好处

用宏来实现链表，主要是可以实现原生C语言中不支持的泛型效果。

> C++ 中可以使用 `stack<T>` 定义一个类型为 `T` 的栈， Java 中可以使用 `HashMap<K,V>` 定义一个键类型为 `K` 且值类型为 `V` 的哈希表。 这种模式称为泛型，C 语言并不支持泛型，因此需要通过宏另辟蹊径来实现「泛型」。

通过宏定义实现链表，将要创造的结构体类型以宏函数的参数 `type` 表示，这样就可以定义多种数据类型、甚至自定义数据类型（如我们创建的Page结构体类型）的链表。相当于实现了其他语言中的「泛型」。

- 性能差异

时间性能差异

插入操作

双向链表：头部插入 $O(1)$ 尾部插入 $O(n)$ 指定节点前 $O(1)$ 指定节点后$O(1)$

单向链表：头部插入 $O(1)$ 尾部插入 $O(n)$ 指定节点前 $O(n)$ 指定节点后$O(1)$

循环链表：头部插入 $O(1)$ 尾部插入 $O(1)$ 指定节点前 $O(1)$ 指定节点后$O(1)$

删除操作

双向链表：头部节点删除 $O(1)$ 尾部节点删除 $O(n)$ 指定节点删除 $O(1)$

单向链表：头部节点删除 $O(1)$ 尾部节点删除 $O(n)$ 指定节点删除 $O(n)$

循环链表：头部节点删除 $O(1)$ 尾部节点删除 $O(1)$ 指定节点删除 $O(1)$

空间性能分析

双向链表：头指针，节点内部双指针+节点数据

单向链表：头指针，节点内部单指针+节点数据

循环链表：头指针，尾指针，节点内部双指针+节点数据

总体时间性能上，循环链表 > 双向链表 > 单向链表

当然空间性能上，循环链表 < 双向链表 < 单向链表

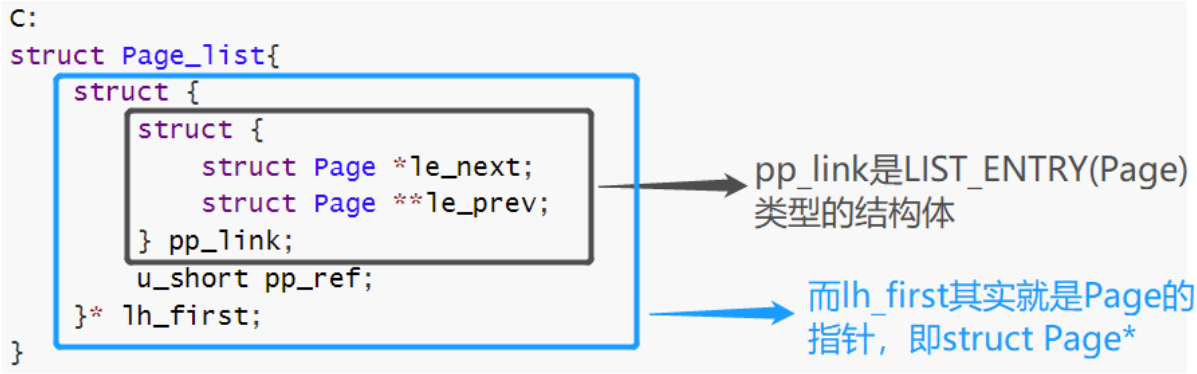
综合实际需求，使用双向链表最佳。

Thinking 2.3

请阅读 `include/queue.h` 以及 `include/pmap.h`，将 `Page_list` 的结构梳理清楚，选择正确的展开结构。

（选项在指导书上，这里就不浪费篇幅了）

MyAnswer



这里结合指导书的相关部分，对 `LIST_ENTRY(type)` 这个宏做一次彻底的解读。

> `LIST_ENTRY(type)` ，作为一个特殊的「类型」出现，它的本质是一个链表项，包括指向下一个元素的指针 `le_next` ，以及指向前一个元素链表项 `le_next` 的指针 `le_prev` 。
>
> `Page_LIST_entry_t` 定义为 `LIST_ENTRY(Page)` ，因此 `pp_link` 即为对应的链表项。

那么，不难看出， `LIST_ENTRY(type)` 实际上就是根据传入的type参数，创造一个包含两个指针的结构体。而这两个指针的类型和名称分别是：

type* le_next

type** le_prev

因此，当传入的type参数是Page时，由于Page本身是一个结构体，那么相应的，上面图片中的黑框部分就应该是那样。le_next是struct Page*类型， le_prev是struct Page**类型，两者加起来构成一个**叫做链表项的结构体**。这里具体的名称则是pp_link。

也就是说，其实pp_link就是由上面的 `LIST_ENTRY(Page)` 生成的。

```
typedef LIST_ENTRY(Page) Page_LIST_entry_t;

struct Page {
    Page_LIST_entry_t pp_link;    /* free list link */

    // Ref is the count of pointers (usually in page table entries)
    // to this page.  This only holds for pages allocated using
    // page_alloc.  Pages allocated at boot time using pmap.c's "alloc"
    // do not have valid reference count fields.

    u_short pp_ref;
};
```

然后，根据指导书的描述，这个链表项应该和其他必要的信息属性data（这里是 `pp_ref` ， `pp_ref` 对应这一页物理内存被引用的次数，它等于有多少虚拟页映射到该物理页。），一起组成链表中每个元素的type类型。而我们惊喜地发现，现在，身为链表项的pp_link和身为data的pp_ref恰好组成了struct Page的结构（如上图）。这就验证了， `LIST_ENTRY(type)` 中的type和 `LIST_NEXT(elm, field)` 等链表操作宏函数中的type是一样的。

因为这些链表操作函数中的参数 elm 和 listelm 都是链表中每个元素的指针，也就是type*类型，并且一定和le_next是同类型（代码中有直接的相互赋值的语句），而根据上面的分析，le_next正是type*类型。而含有le_next和le_prev两个指针的链表项和data部分组成了type结构体。也就是说，下面图中，蓝色的部分的类型和每一个代表每个链表元素的黑框的类型是一样的，都是type*

而Page_list就是图中的头部结构体Head。

结合 `LIST_HEAD(name, type)` 的解释：

> `LIST_HEAD(name, type)` 创建一个名称为 `name` 链表的头部结构体， 包含一个指向 `type` 类型结构体的指针，这个指针可以指向链表的首个元素。

那么Page_list很可能就是由语句 `LIST_HEAD(Page_list, Page*)` 生成。

此外，再说说另一个很重要的链表操作宏函数 `LIST_NEXT(elm, field)` 。摘录指导书内容：

> `LIST_NEXT(elm, field)` ，结构体 `elm` 包含的名为 `field` 的数据，类型是一个链表项 `LIST_ENTRY(type)` ， 返回其指向的下一个元素。下面出现的 `field` 含义均和此相同。

参数elm的类型是type*，而field正是我们前面说了很久的链表项 `LIST_ENTRY(type)` ，那么，“返回链表项指向的下一个元素”其实也就是通过链表项中的le_next指针，返回链表中elm所指向元素的下一个元素的地址（更准确的来说是指向该 下一个元素 的指针），那么自然也是type*类型的。

发现这一点有什么用呢?

1. 我们不难想到：既然宏函数 `LIST_NEXT(elm, field)` 的第一个参数elm和 `LIST_NEXT(elm, field)` 的返回值都是type*类型，那么就可以通过递归嵌套的方式去一直向后查询链表元素。即

```
LIST_NEXT((LIST_NEXT((elm), field)), field); // 这样可以查到当前元素elm的下一个元素的下一个元素
```

这在宏函数 `LIST_INSERT_TAIL(head, elm, field)` 的实现上起了至关重要的作用。

2. 宏函数的返回值类型为type*，那么在宏函数前加一个&，就可以获得type**类型的数据。这样就和 le_prev 指针存储的值 和 le_next 指针自身的值联系了起来。

```
listelm->field.le_next = LIST_NEXT(elm, field);
*(listelm->field.le_next) = &LIST_NEXT(elm, field);
```

Thinking 2.4

请你寻找上述两个 boot_* 函数在何处被调用。

```
static Pte *boot_pgdir_walk(Pde *pgdir, u_long va, int create)
```

和

```
void boot_map_segment(Pde *pgdir,
                      u_long va, u_long size,
                      u_long pa, int perm)
```

MyAnswer

boot_pgdir_walk()这个函数在pmap.c中被调用了1次，在boot_map_segment()函数的定义过程中使用。

```
pgtable_entry = boot_pgdir_walk(pgdir, va_temp, 1);
```

这里是用于拿到一级页表基地址 pgdir 对应的两级页表结构中，va_temp 这个虚拟地址所在的二级页表项的地址。而va_temp又是va + i*BY2PG，相当于实现了按页映射。

boot_map_segment()这个函数在pmap.c中被调用了2次，都是在mips_vm_init()函数中。

```
boot_map_segment(pgdir, UPAGES, n, PADDR(pages), PTE_R);
```

使用 alloc 函数为物理内存管理所用到的 Page 结构体按页分配物理内存，设 npage 个 Page 结构体的大小为 n，一页的大小为 m，由上述函数分析可知分配的大小为 [n/m]*m，同时将分配的空间映射到上述的内核页表 pgdir 中，对应的起始虚拟地址为 UPAGES。

```
boot_map_segment(pgdir, UENVS, n, PADDR(envs), PTE_R);
```

为进程管理所用到的 Env 结构体按页分配物理内存，设 NENV 是最大进程数，NENV 个 Env 结构体的大小为 n，一页的大小为 m，由上述函数分析可知分配的大小为[n/m]*m，同时将分配的空间映射到上述的内核页表 pgdir 中，对应的起始虚拟地址为 UENVS。

Thinking 2.5

请你思考下述两个问题：

- **请阅读上面有关 R3000-TLB 的叙述，从虚拟内存的实现角度，阐述 ASID 的必要性**
- **请阅读《IDT R30xx Family Software Reference Manual》的 Chapter 6，结合 ASID 段的位数，说明 R3000 中可容纳不同的地址空间的最大数量**

MyAnswer

- ASID 的必要性

> 在多任务系统中，所有用户级任务（user-level tasks）都在同一类程序地址（the same sort of program addresses）上执行是很常见的（当然，它们使用的是不同的物理地址）；据说他们使用不同的地址空间（address spaces）。因此，不同任务的映射记录通常会共享相同的“VPN”值。

VPN（Virtual page number，虚拟页码）只是一个低12位被切断的虚拟程序地址（program addresses），因为这低12位不参与翻译过程。

ASID（address space identifier）是用于区分不同进程的一个标识符，因为操作系统可以同时运行多个进程，要是不用 ASID 的话，当操作系统从一个进程切换到另一个进程时，它必须查找与旧进程地址空间相关的所有TLB转换并使其无效，以防止它们被错误地用于新任务。也就是说 TLB 里的所有内容都需要失效（因为进程切换就以为着虚拟地址和物理地址的映射关系切换），而这样是低效的，因为每次 TLB 中的内容清空，就意味着会发生 64 次的冷缺失。

相反，操作系统为每个任务的不同地址空间分配一个6位的唯一代码。在正常运行期间，该代码保存在EntryHi寄存器的ASID字段中，并与程序地址一起使用以形成查找键；因此，与ASID代码不匹配的映射被悄悄地忽略了。这样，即使有不和当前新进程匹配的TLB页表项也不会被立即清除。

- R3000 中可容纳不同的地址空间的最大数量

> 由于ASID只有6位长，如果并发使用的地址空间（address space）超过64个，操作系统软件就必须伸出援手；但这种情况可能不会经常发生。在这样一个系统中，新任务被分配给新的ASID，直到所有64个ASID都被分配；此时，所有任务都将刷新其ASID“取消分配”，并刷新TLB；在重新输入每个任务时，会给出一个新的ASID。因此，ASID冲洗相对较少。

因此，这里R3000 中可容纳不同的地址空间的最大数量为$2^6 = 64$。

Thinking 2.6

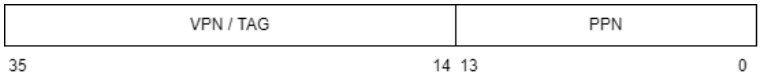
- **tlb_invalidate 和 tlb_out 的调用关系是怎样的?**
- **请用一句话概括 tlb_invalidate 的作用**
- **逐行解释 tlb_out 中的汇编代码**

MyAnswer

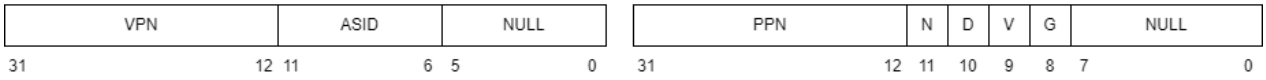
首先附上tlb_invalidate函数的代码，以及Env是进程管理用到的结构体。invalidate是动词“使无效”的意思。这个函数在lab2中出现在page_insert()和page_remove()中。

tlb_invalidate()

GET_ENV_ASID(e->env_id)是取出envid的ASID部分后再右移6位，正好对应了entryHi中低6位是NULL。
PTE_ADDR(pte) 则是将pte的低12位置零。
那么 PTE_ADDR(va) | GET_ENV_ASID(curenv->env_id)便是恰好拼成了一个entryHi的域。



朴素版



真实版

```
#define GET_ENV_ASID(envid) (((envid)>> 11)<<6) // 就是envid中的ASID部分左移6位，现在是12位
// GET_ENV_ASID(e->env_id)是取出envid的ASID部分后再右移6位，正好对应了entryHi中低6位是NULL
#define PTE_ADDR(pte)   ((u_long)(pte)&~0xFFF)  // 将低12位置零
// 那么 PTE_ADDR(va) | GET_ENV_ASID(curenv->env_id)
void tlb_invalidate(Pde *pgdir, u_long va)
{
    if (curenv) {
        tlb_out(PTE_ADDR(va) | GET_ENV_ASID(curenv->env_id));
    } else {
        tlb_out(PTE_ADDR(va));
    }
}
```

可以看出，当curenv不为NULL的时候，此时将va低12位置零后 和 当前进程的ASID 进行或运算后，结果对应的tlb表项的64位置零。否则，直接置零va对应的tlb表项。

然后是一句话概括 tlb_invalidate的作用：**给定特定进程的 context ， tlb_invalidate 函数可以让该进程对应页表中的这个虚拟地址 va 对应的 tlb 项失效。**当然，具体是通过tlb_out()函数实现的。

这里需要详细解读一下tlb_out()函数。

tlb_out()

这个函数主要实现的是给定一个 tlb 需要的键值 Key（即表项的高 32 位，包括 VPN 和 ASID），把这个键值对应的表项置 0 （就是 64 位全是 0）。

首先要说的是，汇编函数调用的时候，传入的参数默认进入$a0-$a4寄存器，超过4个的时候进入栈。因此在这里，**无论是 `PTE_ADDR(va) | GET_ENV_ASID(curenv->env_id)` 还是 `PTE_ADDR(va)` ，他们都是进入了$a0寄存器。**而$a0寄存器在程序中的宏声明是 `CP0_CONTEXT` ，所以会用context指代传入的参数。

在协处理器里面与 tlb 有关的寄存器如下表：

寄存器	编号	作用
EntryHi	10	保存某个 tlb 表项的高 32 位，任何对 tlb 的读写，都需要通过 EntryHi 和 EntryLo。重点关注其与TLB的协作关系, 以及中断异常发生时该寄存器的行为
EntryLo	2	保存某个 tlb 表项的低 32 位。重点关注其与TLB的协作关系
Index	0	决定索引号为某个值的 tlb 表项被读或者写
Random	1	提供一个随机的索引号用于 tlb 的读写

相应的宏定义

```
#define CP0_INDEX $0
#define CP0_RANDOM $1
#define CP0_ENTRYLO0 $2
#define CP0_ENTRYLO1 $3
#define CP0_CONTEXT $4      // $a0 这里就是context
#define CP0_ENTRYHI $10
#define CP0_ECC $26         // $k0
#define CP0_CACHEERR $27    // $k1
// k0 k1 这两个寄存器为中断异常处理函数保留，用户态中不应该使用这两个寄存器.
```

然后，介绍和CP0协处理器有关的两个MIPS指令。

mfc0 (Move from Coprocessor 0) To move the contents of a coprocessor 0 register to a general register.

mtc0 (Move to Coprocessor 0) To move the contents of a general register to a coprocessor 0 register.

接着是与 tlb 相关的MIPS指令。

指令	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。
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	tlb probe。用于查看 tlb 是否可以转换虚拟地址（即命中与否）。根据 EntryHi 中的 Key（包含 VPN 与 ASID），查找 TLB 中与之对应的表项。如果命中，将表项的索引存入 Index 寄存器**中的Index(8~13位)，恰好是6位，$2^6 = 64$项，刚好是TLB的表项数目**。若未找到匹配项，则 Index 最高位被置 1（相当于是负数）。

最后我们来看这个函数的实现

```
LEAF(tlb_out)
//1: j 1b
nop
```

```
    mfc0    k1,CP0_ENTRYHI     # 首先把原来 EntryHi 寄存器里的数据保存下来 （放入k1）
                               # 因为一会儿要把我们的键值放进去，怕数据被覆盖
    mtc0    a0,CP0_ENTRYHI     # 把tlb_out收到的参数键值移入 EntryHi
    nop
    // insert tlbp or tlbwi
     tlbp                      # 查询，将查询的结果存入 Index 寄存器
    nop
    nop
    nop
    nop
    mfc0    k0,CP0_INDEX       # 把 Index 的内容移出来（Index中保持着查询结果）
    bltz    k0,NOFOUND         # 通过将Index中的结果与 0 比较，判断有没有查到，如果没查到
Index最高位被置1，就会小于0
    nop
    # 如果查到了，在进行NOFOUND里的操作之前，还要将这个TLB条目全部清零
    mtc0    zero,CP0_ENTRYHI   # 将 EntryHi 清零
    mtc0    zero,CP0_ENTRYLO0 # 将 EntryLo 清零
    nop
    // insert tlbp or tlbwi
     tlbwi                     # 将 EntryHi 和 EntryLo 的内容写入 index 对应的 tlb条目
                               # 因为现在 EntryHi 和 EntryLo 都是0，就相当于把对应的 tlb
条目 置零
NOFOUND:
    mtc0    k1,CP0_ENTRYHI     # 如果没查到，就只进行：将原来的 EntryHi 恢复
    j   ra                     # 返回
    nop
END(tlb_out)
```

Thinking 2.7

在现代的 64 位系统中，提供了 64 位的字长，但实际上不是 64 位页式存储系统。假设在 64 位系统中采用三级页表机制，页面大小 4KB。由于 64 位系统中字长为 8B，且页目录也占用一页，因此页目录中有 512 个页目录项，因此每级页表都需要 9 位。因此在 64 位系统下，总共需要 3 × 9 + 12 = 39 位就可以实现三级页表机制，并不需要 64 位。现考虑上述 39 位的三级页式存储系统，虚拟地址空间为 512 GB，若记三级页表的基地址为 PTbase ，请你计算：

- **三级页表页目录的基地址**
- **映射到页目录自身的页目录项(自映射)**

MyAnswer

三级映射要经历三步。

首先，虚拟地址（512G）映射到三级页表区（512*512*512=128M个8B页表项，1G大小）。

每个页表项映射了4K的空间，则PTbase对应的页表项是第PTbase>>12个，而每个页表有512个页目录项，每个页表项的大小是8B（页面大小 4KB，有512个页表项），所以PTbase在三级页表区中对应的偏移量是PTbase>>9。记作二级页表项基地址。

然后，三级页表区（1G）映射到二级页表区（512*512=256K个8B页表项，2M大小）。

每个页表项映射了4K的空间，则二级页表区基地址对应的页表项是第(PTbase>>9)>>12个，所以二级页表区基地址在二级页表区中对应的偏移量是(PTbase>>9)>>9。记作一级页表基地址。

最后，二级页表区（2M）映射到一级页表（512个8B页表项，4K大小）。

每个页表项映射了4K的空间，则一级页表基地址对应的页表项是第(PTbase>>18)>>12个，所以一级页表基地址在一级页表自身中对应的偏移量是(PTbase>>18)>>9。

那么，三级页表页目录（即一级页表）的基地址为 PTbase + (PTbase>>9) + (PTbase>>18)

映射到页目录自身的页目录项是 PTbase + (PTbase>>9) + (PTbase>>18) + (PTbase>>27)

> PS:其实在不考虑页表项是否能整除的前提下，直接按照映射两者的大小做比例可能更快一点

记录一个大胆的猜想：多级页表之间始终实现的是，高一级的一个页表项代表低一级的一个页表，所以每一个页表的大小之所以和虚拟页大小相同，是因为页表在高一级的页表看来，也是“页”。最底层的虚拟页何尝不是一种最低级的页表呢。

Thinking2.8

任选下述二者之一回答：

- **简单了解并叙述 X86 体系结构中的内存管理机制，比较 X86 和 MIPS 在内存管理上的区别。**
- **简单了解并叙述 RISC-V 中的内存管理机制，比较 RISC-V 与 MIPS 在内存管理上的区别。**

MyAnswer

这部分参考刘子楠同学的解读和这篇文章：NFV关键技术：x86架构基础（上篇） - 知乎 (zhihu.com)

x86架构的内存管理机制分为两部分：**分段机制和分页机制**。

> 分段机制为程序提供彼此隔离的代码区域、数据区域、栈区域，从而避免了同一个处理器上运行的多个程序互相影响。
>
> 分页机制实现了传统的按需分页、虚拟内存机制，可以将程序的执行环境按需映射到物理内存。此外，分页机制还可以用于提供多任务的隔离。
>
> 分段机制和分页机制都可以通过配置，支持简单的单任务系统、多任务系统或共享内存的多处理器系统。**需要强调的一点是，处理器无论在何种运行模式下都不可以禁止分段机制，但是分页机制却是可选选项。**

x86完成了分段和分页的结合

- 分段：**完成了虚拟地址到线性地址的映射**

  x86架构提供了两种段描述符表：**GDT（全局段描述符表Global Descriptor Table）**和**LDT（本地段描述符表Local Descriptor Table）**。其中GDT描述系统段，包括操作系统本身；LDT描述局部于每个系统的段，包括其代码、数据、堆栈等。

  为了访问一个段，一个x86程序必须把这个段的**选择子**装入机器的6个段寄存器中的某一个中，对应的描述符从GDT和LDT取出装入微程序寄存器中。

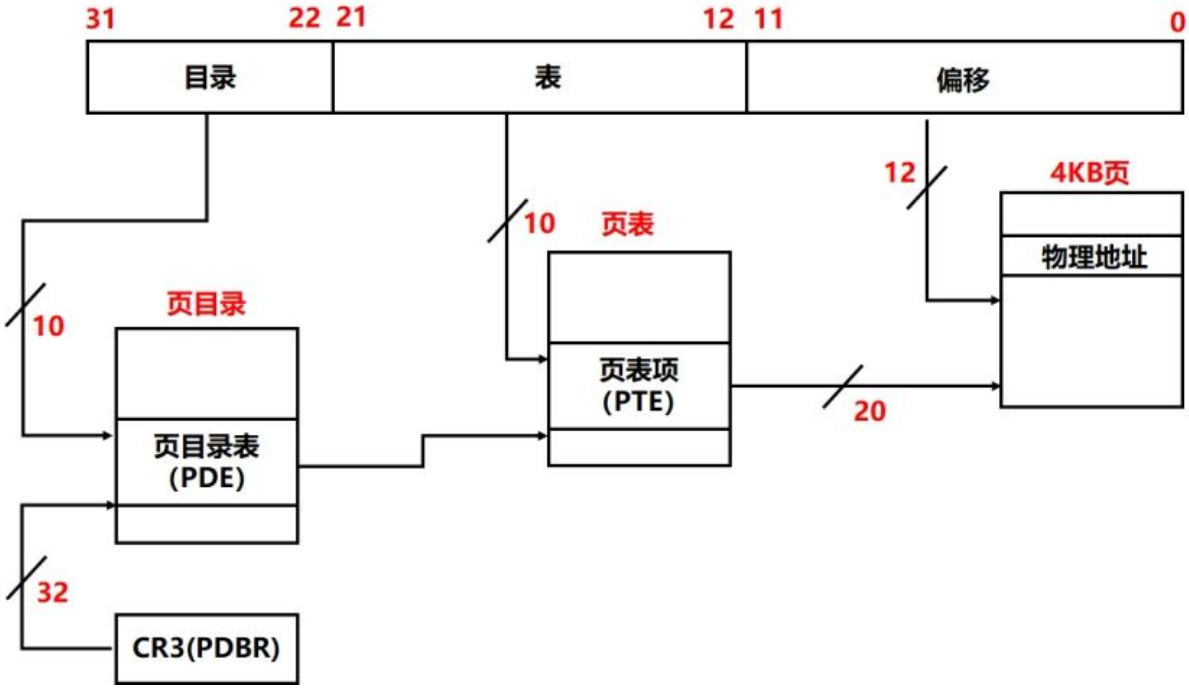
  根据描述符可以计算得到线性地址。

- 分页：**完成了线性地址到物理地址的映射**

  > 页表是用于将线性地址转换成物理地址的主要数据结构。一个地址对齐到页边界后的值称为页帧号（或者页框架），它实际上就是该地址所在页面的基地址。比如：一个页大小为4kB，那么第一个页帧号就是0，第二个页帧号就是4096，依次类推。线性地址对应的页帧号叫做虚拟页帧号（Virtual Frame Number，下面简称为VFN），物理地址对应的页帧号叫做物理页帧号（Physical Frame Number，下面简称为PFN）或机器页帧号。**页表实际上是存储VFN到PFN映射的数据结构。**

  如果禁止分页，则线性地址直接被解释为物理地址。此时相当于纯粹的分段方案。

  如果允许分页，则线性地址就被解释为存储着一级二级两级页表偏移量+页内偏移量。通过一个二级页目录体系，在转换为物理地址后被送往存储器用于读写操作。



> CR3寄存器也称为**页目录基地址寄存器**（Page-Directory Base Register，PDBR），存放着页目录的物理地址。一个进程在运行前，必须将其页目录的基地址存入CR3，而且，页目录的基地址必须对齐到4KB页边界。启用PAE时，CR3指向页目录指针表，每一项都指向一个页目录表，共有4个页目录表。

$Part2$ 实验难点图示

这个实验最大的难点应该还是初次接触一个比之前都复杂的项目结构，如何弄清楚这个项目到底在干什么，总体架构是什么样。下面我在这一部分想梳理一遍整个实验到底做了些什么。

$Part2.1$ 物理内存管理部分

细品指导书

内存控制块

物理内存管理，MOS 中 维护 `npage` 个内存控制块，也就是 `Page` 结构体。每一个内存控制块对应一页的物理内存，注意并不是这个结构体就是对应的物理内存，而是用这个结构体来管理这块内存的分配。

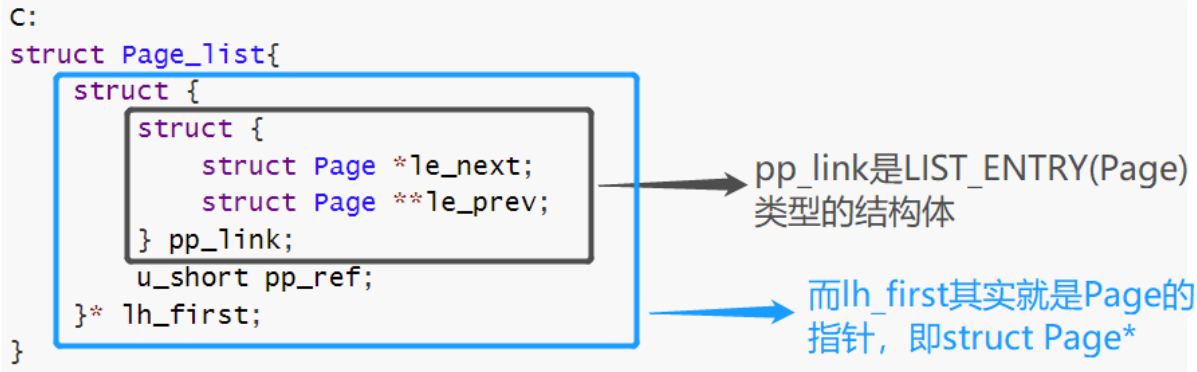
`npage` 个 `Page` 和 `npage` 个物理页面一一顺序对应，具体来说， `npage` 个 `Page` 的起始地址为 `pages` ，则 `pages[i]` 对应从 `0` 开始计数的第 `i` 个物理页面。两者的转换可以使用 `include/pmap.h` 中的 `page2pa` 和 `pa2page` 这两个函数。

将这些结构体全部插入一个链表中，这个链表被称为空闲链表，它对应 `page_free_list` 。

当一个进程需要被分配内存时，将链表头部的内存控制块对应的那一页物理内存分配出去，同时将该内存控制块从空闲链表的头部删去。

当一页物理内存被使用完毕 (准确来说，引用次数为 `0` 时)，将其对应的内存控制块重新插入到空闲链表的头部。

`pp_ref` 对应这一页物理内存被引用的次数，它等于有多少虚拟页映射到该物理页。



接下来需要搞清楚，物理内存控制块和物理内存之间的映射关系

在评测姬中打印sizeof(struct Page)，可以得到struct Page类型是12个字节。也就是说，一个物理内存控制块的大小就是12 B，而要管理的物理页面的大小是4K。

相关函数详细解析

mips_detect_memory()

`mips_detect_memory()`，实现位于 mm/pmap.c，它的作用是对一些和内存管理相关的变量进行初始化，包括：

- `maxpa`，它的含义是物理地址的最大值 +1，即 `[0, maxpa - 1]` 范围内所有整数所组成的集合等于物理地址的集合。
- `basemem`，表示物理内存对应的字节数。
- `npage`，表示**总物理页数**。
- `extmem`，表示扩展内存的大小，本实验中不涉及扩展内存，直接设为 0 即可。

这个函数本身很简单，但是却确定了一些贯穿实验的常量定义。

MOS中物理内存大小是64MB，即maxpa = 2^26 = 0x400 0000

MOS中页的大小是4KB，即贯穿整个实验的常量BY2PG = 4096

在本实验中，extmem是0，所以basemem就是0x400 0000。那么**npage = basemem/BY2PG = 2^(26-12) = 2^14 = 0x4000** 。

alloc()

```
static void *alloc(u_int n, u_int align, int clear){
      extern char end[];
      u_long alloced_mem;
      /* Step 1: Round up `freemem` up to be aligned properly */
      /* Step 2: Save current value of `freemem` as allocated chunk. */
      /* Step 3: Increase `freemem` to record allocation. */
      /* Check if we're out of memory. If we are, PANIC !! */
      /* Step 4: Clear allocated chunk if parameter `clear` is set. */
      /* Step 5: return allocated chunk. */
  }
```

alloc函数的作用是：分配 n 字节的空间并返回初始虚拟地址 `(void *)alloced_mem`，同时保证 align 可以整除函数返回的初始虚拟地址，若 clear 为真则将这次分配的空间的值清零，否则不清零。

具体代码解析指导书已经很详细了，这里只是做几点补充。

> `extern char end[];` 显然是一个定义在该文件之外的变量，它并不在一个 C 代码文件中，而是在 lab1 填写的 tools/scse0_3.lds 中：

```
. = 0x80400000;
end = . ;
```

也就是说它对应虚拟地址 0x80400000 ，根据映射规则，对应的物理地址是 0x00400000 ，因此我们所管理的物理地址区间为 [0x00400000, maxpa – 1] 。同时，这段物理地址区间和虚拟地址区间 [0x80400000, 0x83ffffff] 一一对应，因此当虚拟地址被分配出去，代表对应的物理地址也被分配了出去。

这里主要是结合上一次实验内核部分的图做一个宏观感知：

然后，这里的映射规则是什么呢？其实在两个宏里面就可以看出。

两个非常重要的宏PADDR 和 KADDR。完成物理地址和虚拟地址的转化。

```
// translates from kernel virtual address to physical address.
#define PADDR(kva)                  \
   ({                         \
      u_long a = (u_long) (kva);          \
      if (a < ULIM)              \
         panic("PADDR called with invalid kva %08lx", a);\
      a - ULIM;                  \
   })
// translates from physical address to kernel virtual address.
#define KADDR(pa)                   \
   ({                         \
      u_long ppn = PPN(pa);             \
      if (ppn >= npage)             \
         panic("KADDR called with invalid pa %08lx", (u_long)pa);\
      (pa) + ULIM;               \
   })
```

里面用到的宏定义为

```
#define ULIM 0x80000000
```

那么，这两个宏反应的就是：虚拟->物理，减去0x8000 0000；物理->虚拟，加上0x8000 0000；正好是[0x80400000, 0x83ffffff] 到 [0x00400000, maxpa – 1] 的映射，也就是**kseg0的转换规则：最高1位置零。**

下面就是对这个函数的解读：

- Step 0，初始化 `freemem`

```
/* Initialize `freemem` if this is the first time.
 * The first virtual address that the linker did *not* assign
 * to any kernel code or global variables.
 */
if (freemem == 0) {
    freemem = (u_long)end;
}
```

> `freemem` 作为一个全局变量，初始值为 0，因此第一次调用该函数将其赋为 `end`， `freemem` 的含义是 `[0x80400000, freemem - 1]` 的虚拟地址都已经被分配了。

- Step 1， `ROUND(a, n)` 是一个定义在 include/types.h 的宏，它的作用是返回 `[a/n]*n` （这里的中括号表示向上取整），要求 `n` 必须是 2 的非负整数次幂，因此 `align` 也必须符合。这行代码的含义即找到**最小的符合条件的初始虚拟地址** `freemem`，中间未用到的地址空间全部放弃。

```
freemem = ROUND(freemem, align);
```

- Step 2，将 `alloced_mem` 设为初始虚拟地址。

```
alloced_mem = freemem;
```

- Step 3，将 `freemem` 增加 `n`，表示这一段都被使用了。 `PADDR(x)` 是一个返回虚拟地址 `x` 所对应物理地址的宏，它定义在 include/mmu.h，这里要求 `x` 必须**是 kseg0 区域内的虚拟地址**，这部分的虚拟地址只需要将二进制下最高位清零就可以得到对应的物理地址。这段代码的含义即检查分配的空间是否超出了最大物理地址，若是则产生报错。

```
freemem = freemem + n;
if (PADDR(freemem) >= maxpa) {
    panic("out of memory\n");
    return (void *)-E_NO_MEM;
}
```

- Step 4，如果 `clear` 为真，则使用 `bzero` 函数将这一部分清零， `bzero` 函数的实现位于 init/init.c，实现方法类似于 `bcopy` 函数。

```
if (clear)
    bzero((void *)alloced_mem, n);
```

- Step 5，最后将初始虚拟地址返回即可。

```
return (void *)alloced_mem;
```

bzero()

将 void*b 代表的地址开始的len个字节清零，在之后的使用中，一定要牢记，第一个参数是虚拟地址。因为本身bzero()处理的就是虚拟地址。

```
void bzero(void *b, size_t len)
{
    void *max;
    max = b + len;
    //printf("init.c:\tzero from %x to %x\n",(int)b,(int)max);
    // zero machine words while possible
    while (b + 3 < max) {
        *(int *)b = 0;
        b += 4;
    }
    // finish remaining 0-3 bytes
    while (b < max) {
        *(char *)b++ = 0;
    }
}
```

bcopy()

```
void bcopy(const void *src, void *dst, size_t len)
{
    void *max;
    max = dst + len;
    // copy machine words while possible
    while (dst + 3 < max) {
        *(int *)dst = *(int *)src;
        dst += 4;
        src += 4;
    }
    // finish remaining 0-3 bytes
    while (dst < max) {
        *(char *)dst = *(char *)src;
        dst += 1;
        src += 1;
    }
}
```

mips_vm_init()

- 使用 `alloc` 函数为内核一级页表分配 1 页物理内存，并将 `boot_pgdir` 设为对应的初始虚拟地址。
- 使用 `alloc` 函数为物理内存管理所用到的 `Page` 结构体按页分配物理内存，设 `npage` 个 `Page` 结构体的大小为 `n`，一页的大小为 `m`，由上述函数分析可知分配的大小为 `[n/m]*m`，同时将分配的空间映射到上述的内核页表中，对应的起始虚拟地址为 `UPAGES`

这里UPAGES是0x8000 0000 - 2*PDMAP 得到的

- 为进程管理所用到的 Env 结构体按页分配物理内存，设 NENV 是最大进程数，NENV 个 Env 结构体的大小为 n，一页的大小为 m，由上述函数分析可知分配的大小为[n/m]*m，同时将分配的空间映射到上述的内核页表中，对应的起始虚拟地址为 UENVS。这一部分涉及 lab3 知识，不过可以看出它和上一条几乎是一样的，仅有的区别在于结构体不同和对应结构体的数量不同。上述映射用到了 `boot_map_segment` 函数，它的具体作用会在后面详细叙述。

这里UENVS是0x8000 0000 - 3*PDMAP 得到的

```
void mips_vm_init()
{
    extern char end[];
    extern int mCONTEXT;
    extern struct Env *envs;
    Pde *pgdir;
    u_int n;
    /* Step 1: Allocate a page for page directory(first level page table). */
    // 为一级页表分配一页内存
    pgdir = alloc(BY2PG, BY2PG, 1);
    printf("to memory %x for struct page directory.\n", freemem);
    // 上面这行代码的正常执行结果是 to memory 80401000 for struct page directory
    // 说明一页内存大小就是0x8040 1000 - 0x8040 0000 = 0x1000 = 2^12 = 4KB
    mCONTEXT = (ixnt)pgdir;

    boot_pgdir = pgdir;// Pde *boot_pgdir 是定义在pmap.c的一个全局指针变量

    /* Step 2: Allocate proper size of physical memory for global array `pages`,
     * for physical memory management. Then, map virtual address `UPAGES` to
     * physical address `pages` allocated before. In consideration of alignment,
     * you should round up the memory size before map. */
    pages = (struct Page *)alloc(npage * sizeof(struct Page), BY2PG, 1);
    printf("to memory %x for struct Pages.\n", freemem);
    // 上面这行代码的正常执行结果是 to memory 80431000 for struct Pages.
```

```
    // 说明总共内存控制块分配得到的大小是 0x3 0000 =3*2^16 = 3*64KB
    // 而总共有2^14个物理页面，每个内存控制块是struct Page类型，也就是12字节（这个在评测姬验证过了），
    // 则内存控制块总大小为n = 2^14 * 12 = 3*2^16，因为是4KB的整数倍，所以和分配的到的大小是吻合的
    // 内存控制块总共需要3*2^16 / 2^12 = 3*2^4 = 48页

    n = ROUND(npage * sizeof(struct Page), BY2PG);
    boot_map_segment(pgdir, UPAGES, n, PADDR(pages), PTE_R);
    // 结合boot_map_segment函数，这里就很清晰了
    // 将一级页表基地址 pgdir 对应的两级页表结构做区间地址映射，将虚拟地址区间 [UPAGES, UPAGES+n-1] 映射到物理地址区间 [PADDR(pages), PADDR(pages)+n-1]，因为是按页映射，要求 n 必须是页面大小的整数倍。同时为相关页表项的权限为设置为PTE_R，即可写。
    // 这里不难看出 只要调用boot_map_segment函数，就要先用ROUND把区间大小size调整为页面整数倍

    printf("now freemem is %x.\n", freemem); // 这行代码是自己加的
    // 上面这行代码的运行结果是 now freemem is 80432000. 说明映射过程申请了一页二级页表

    /* Step 3, Allocate proper size of physical memory for global array `envs`,
     * for process management. Then map the physical address to `UENVS`. */

    envs = (struct Env *)alloc(NENV * sizeof(struct Env), BY2PG, 1);
    printf("to memory %x for struct Envs.\n", freemem); // 这行代码是自己加的
    // 上面这行代码的正常执行结果是 to memory 8046c000 for struct Envs.
    // 说明总共进程控制块分配得到的大小是 0x8046 c000 - 0x8043 2000 = 0x3 a000 = 29*2^13 B
    // 总共有NENV(2^10)个进程，每个进程控制块结构体struct Env的大小是216字节（这个在评测姬验证过了），但是在lab3里面又变成了232字节，所以说
    // 则进程控制块的总大小为n = 2^10 * 232 = 29*2^13 B，因为是4KB的整数倍，所以和分配的到的大小是吻合的。进程控制块总共占用  29*2^13 / 2^12 = 58页
    n = ROUND(NENV * sizeof(struct Env), BY2PG);
    boot_map_segment(pgdir, UENVS, n, PADDR(envs), PTE_R);

    printf("now freemem is %x.\n", freemem); // 这行代码是自己加的
    // 上面这行代码的运行结果是 now freemem is 8046d000.
    printf("pmap.c:\t mips vm init success\n");
}
232 = 8 * 29
```

整体梳理内存控制块的映射过程（极度重要）

首先，我们的意图是用boot_map_segment()函数，将一级页表基地址 pgdir 对应的两级页表结构做区间地址映射，将虚拟地址区间 [UPAGES, UPAGES+n−1] 映射到物理地址区间 [PADDR(pages), PADDR(pages)+n−1]。其中，n就是物理内存控制块的总大小3*2^16 B。因为是按页分配，实际上只占用了48页。

然后在boot_map_segment()函数中，直接只在这个函数里做的部分，其实只有循环48次，在对应的48个二级页表项中写入希望映射到的物理地址。并设置必要的权限位置。

```
pgtable_entry = boot_pgdir_walk(pgdir, va_temp, 1);
*pgtable_entry = (pa + i) | perm | PTE_V;
```

而取得二级页表项的地址的工作，则是由每次循环中的boot_pgdir_walk()函数做到。由于只取48次，对应一级页表项根本没有变化（基地址pgdir+PDX(va)，但是48个va的22~31位始终相同）。而这一个一级页表项一开始并不存在对应的二级页表（因为pgdir整个都是0），所以需要额外申请一页二级页表。而一旦申请完毕，经过计算，PTX(va)是0到47的48个整数，那么就以&pgtable[PTX(va)]直接返回二级页表项地址即可。

具体计算参见上面的两个函数部分

总的来说，达到了这样的效果：进程 用boot_pgdir中 偏移量分别是PDX(UPAGES)和PDX(UENVS)的两个一级页表项 找到各自的二级页表，然后利用这两个二级页表的信息去找到对应的物理内存控制块和进程控制块。

在Page的二级页表中，存放的是内存控制块的对应的物理页的物理地址，48个页表项，相邻的页表项之间相差4K个物理内存控制块，则对应的物理地址相差4K*4K。

这样，在lab6中，其中，UPAGES对应着二级页表中的第一项，也就是物理内存控制块所对应物理页的物理地址的起点。这样形成了一个UPAGE (0x7f80 0000) -- pages[0] (0x8040 1000) --- 0x0000 1000(物理内存)的映射了

struct Page *upages=(struct Page*) UPAGES;
用户态
upages[i]
等价于内核态
pages[i]

此外，通过自己改变这个函数，在这个函数中输出 struct Page的大小，得到是c，即12个字节。而u_short的大小是2个字节，struct Page*和struct Page**都是4个字节，结合struct Page的结构图，可以得出，我们的分析是正确的，因为u_short要在结构体中对齐，所以也是占用4个字节，一共12字节。

```
typedef LIST_ENTRY(Page) Page_LIST_entry_t;

struct Page {
    Page_LIST_entry_t pp_link;    /* free list link */

    // Ref is the count of pointers (usually in page table entries)
    // to this page.  This only holds for pages allocated using
    // page_alloc.  Pages allocated at boot time using pmap.c's "alloc"
    // do not have valid reference count fields.

    u_short pp_ref;
};
```

page_init()

```
void page_init(void)
{
    /* Step 1: Initialize page_free_list. */
    /* Hint: Use macro `LIST_INIT` defined in include/queue.h. */
    LIST_INIT(&page_free_list);
    /* Step 2: Align `freemem` up to multiple of BY2PG. */
    freemem = ROUND(freemem, BY2PG);
    /* Step 3: Mark all memory blow `freemem` as used(set `pp_ref`
     * filed to 1) */
    int i = 0;
    int unavailble_npage = PPN(PADDR(freemem)); // PPN(x) <==> x>>12 <==> x / BY2PG
    for (i = 0; i < unavailble_npage; i++){
        pages[i].pp_ref = 1;
    }
    /* Step 4: Mark the other memory as free. */
    for (i = unavailble_npage; i < npage; i++) {
        pages[i].pp_ref = 0;
        LIST_INSERT_HEAD(&page_free_list, &pages[i], pp_link);
    }
```

```
    LIST_REMOVE( pa2page(PADDR(TIMESTACK - BY2PG)), pp_link );// lab3新增
}
```

这里主要是注意PPN这个宏函数。无论是PPN还是VPN，都相当于直接除以每一页的大小（4KB），这样可以算出当前地址va（无论是物理地址还是虚拟地址）地址对应的到底有多少页。

```
// page number field of address
#define PPN(va)       (((u_long)(va))>>12)
#define VPN(va)       PPN(va)
```

以及，在lab3中，要求从page_free_list中删去TIMESTACK - BY2PG对应的一页

page_alloc() *极度重要

```
int page_alloc(struct Page **pp)
{
    struct Page *ppage_temp;
    /* Step 1: Get a page from free memory. If fail, return the error code.*/
    if (LIST_EMPTY(&page_free_list)){/* I. `page_free_list` is empty */
        return -E_NO_MEM;
    }
    ppage_temp = LIST_FIRST(&page_free_list);/* II. the first item in
`page_free_list` */
    LIST_REMOVE(ppage_temp, pp_link);/* III. remove this page from the list */
    /* Step 2: Initialize this page.
     * Hint: use `bzero`. */
    bzero((void *)page2kva(ppage_temp), BY2PG);/* IV. kernel virtual address of
this page */
    // 这里假设ppage_temp - pages = n，那么 n<<12 的结果的最高位置1就是虚拟地址
    *pp = ppage_temp;
    return 0;
    // zero indicates success.
}
```

这里主要写一些对于这个函数的理解。

> 这个函数主要是实现从空闲的内存中申请一页物理内存，但并不是调用alloc函数，而是通过在之前维护好的page_free_list来实现。

这里用更易懂的话解释就是：“申请物理内存”具体表现为是申请一个对应的物理内存控制块（即返回struct Page结构体的指针），并将这个物理内存控制块从page_free_list中移除。同时，将这个物理内存控制块对应的内核虚拟地址起的那一页虚拟内存置零后，将这页起始地址返回。也就是说，其实返回的还是一页虚拟内存的基地址。

由于mips_vm_init()函数是在page_init()函数之前，在mips_vm_init()早已经分配好了物理内存控制块和进程控制块的虚拟内存地址。而在page_init()函数中建立起了freemem和page_free_list的对应关系，所以在page_free_list中的物理内存控制块所对应的物理地址转换为的虚拟地址一定在执行完mips_vm_init()函数后的freemem（具体来说，是0x8046d000）和0x8400 0000-1之间。所以page_alloc()分配出的虚拟地址不会干扰物理内存控制块本身和进程控制块本身。并且始终在[0x8046 d000, 0x8400 0000-1]之间。

包括在后面的进程相关中，也有如下的应用：

每申请一个进程，都要为它调用一次page_alloc()函数，分配一页内存。这里的内存就是虚拟内存，只不过切切实实对应着一页物理内存。程序中在kseg0分配一页虚拟内存，对应到硬件层面就是分配一页物理内存。

需要为新进程的程序分配空间来容纳程序代码。当需要加载一个ELF文件时，也是把binary文件中各部分的segment利用bcopy()函数，拷贝到page_alloc()函数分配的若干页内核虚拟内存中。

> 感觉“分配一页物理内存”的表述虽然不能说有问题，但是感觉有点误导，因为实际上在MIPS程序里分配的还是一页虚拟内存。只不过这一页虚拟内存切切实实对应着一页物理内存和对应的物理内存控制块。这就是物理内存和虚拟内存的映射的重要性。

具体步骤：

首先，检查page_free_list这个Head类型结构体（只含有一个type*指针）指向的链表是否为空。如果为空的话，说明已经没有空闲内存以供分配，报内存相关的错误。`LIST_EMPTY`宏定义如下。

```
#define LIST_EMPTY(head)        ((head)->lh_first == NULL)
```

如果不为空，就让struct Page *类型的ppage_temp 指向 page_free_list 牵头的链表的第一个元素。

然后，将第一个元素从链表中删去。`LIST_REMOVE`宏定义如下。

```
#define LIST_REMOVE(elm, field) do {                                \
    if (LIST_NEXT((elm), field) != NULL)                            \
        LIST_NEXT((elm), field)->field.le_prev = (elm)->field.le_prev; \
    *((elm)->field.le_prev) = LIST_NEXT((elm), field);              \
} while (0)
```

可以看到，这里其实说明了`LIST_NEXT(elm, field)`和`((elm)->field.le_next)`没区别。以及以`*((elm)->field.le_prev)`方式**取到前一个元素的next指针**。

```
#define LIST_NEXT(elm, field)   ((elm)->field.le_next)
```

然后，将申请的这一页空间置零初始化。

同时，这里需要把struct Page*类型的ppage_temp利用函数`page2kva`转换成虚拟地址类型。关于这个地址转换函数和三种地址的解析会在后面提到。

page_free()

根据传入的物理内存控制块的地址pp，判断控制块对应的物理页面引用次数是否为0，如果是，把这个pp对应的物理内存控制块插进page_free_list。如果引用次数小于0，那么就报错。

```
/*Overview:
  Release a page, mark it as free if it's `pp_ref` reaches 0.
Hint:
When you free a page, just insert it to the page_free_list.*/
void page_free(struct Page *pp)
{
   /* Step 1: If there's still virtual address referring to this page, do
nothing. */
    if (pp->pp_ref > 0){
        return;
    }
   /* Step 2: If the `pp_ref` reaches 0, mark this page as free and return. */
    else if (pp->pp_ref == 0){
        LIST_INSERT_HEAD(&page_free_list, pp, pp_link);
        return;
    }
    else
   /* If the value of `pp_ref` is less than 0, some error must occurr before,
    * so PANIC !!! */
   panic("cgh:pp->pp_ref is less than zero\n");
}
```

详细解析在系统中出现的三种地址

目前，在我们的系统内，一共有三种地址，虚拟地址va，物理地址pa，以及物理内存控制块地址。前两种是u_long类型，后一种是struct Page*类型。**以及物理内存控制块的指针之间做差得到的物理页号。**

但是，值得注意的是，这三种地址都是32位。无论是虚拟地址还是物理控制块地址，在直接转化为物理地址的时候，只要是在kseg0，都可以直接使用PADDR宏，在最高位置0。

物理控制块地址直接转为物理地址

物理控制块地址直接转为物理地址的例子——mips_vm_init()函数中的

```
boot_map_segment(pgdir, UPAGES, n, PADDR(pages), PTE_R);
boot_map_segment(pgdir, UPAGES, n, PADDR(envs), PTE_R);
```

物理控制块地址间接转为物理地址——以物理页数/物理内存控制块号为中介

这是源自使用page2pa带来的迷惑性——这里实际上是另一种物理控制块地址转为物理地址的方式。这里方式通过page2ppn求出这是第几个物理控制块/物理页，再由物理“页数”左移12位。

这里左移12位的不是物理控制块地址，而是物理内存控制块自己的地址减去物理内存控制块数组首地址后的差值——对应着这是第几个物理页。

上面这个值是两个struct Page*类型的指针的减法结果。由于指针减法的特性，减法结果实际上自动除以了struct Page*类型的大小（**两个同类型指针相减的结构不是以字节为单位的,而是以指针指向的数据类型大小为单位的**），得到的就是pages数组的下标索引，反应了它是第几个Page结构体，自然也是第几个物理页面。

因此，这个值就是一个$[0, npage-1] = [0, 2^{14}-1]$之间的数，虽然它也是32位数，但是它的高位全是0，而且有效数字不会超过14位，因此可以将其左移12位得到物理地址。当然，物理内存控制块地址（即struct Page*类型）本身是$pages(\text{数组基地址}) + \text{个数} \times 12(\text{每个结构体大小})$。

MOS中物理内存大小是64MB，则对应地址的范围是0x0000 0000 - 0x0040 0000。正好是除了高位0以外，有效的26位=第几个物理页[不超过14位数]<<12[左移12位]

总的来说：目前操作的物理地址的高位全是0，有效的数字并不够32位。意识到这一点很重要。

所以在这种间接转化下：

物理内存控制块地址 到 物理地址 的转换就是，当前页数的地址减去pages这个数组的首地址以后的结果，通过左移12位得到物理页数对应的物理地址，然后通过KADDR转换为虚拟地址。因为物理页数本身就是页“数”嘛，所以转换成的物理地址一定是页面大小的整数倍。

物理地址 到 物理内存控制块地址 的转换就是，在合法性检查结束以后，将物理地址右移12位得到的结果（即算出它是在哪一页上）作为pages数组的索引。即&pages[PPN(pa)]。

> 为什么要绕一个圈子呢？我想大概是因为间接的方式可以得到这是第几个物理内存控制块，也就是第几个物理页。这个还是比较有用的。

下面是相应的地址转换的函数

```
/* Get the Page struct whose physical address is 'pa'. */
static inline struct Page *pa2page(u_long pa)
{
    if (PPN(pa) >= npage) {
        panic("pa2page called with invalid pa: %x", pa);
    }
    return &pages[PPN(pa)];
}
/* Get the Page struct whose physical address is 'pa'. */
```

```
static inline u_long page2pa(struct Page *pp)
{
    return page2ppn(pp) << PGSHIFT; // PGSHIFT = 12
}
/* Get the kernel virtual address of Page 'pp'. */
static inline u_long page2kva(struct Page *pp)
{
    return KADDR(page2pa(pp));
}
// 得到page地址对应的物理页数
static inline u_long page2ppn(struct Page *pp)
{
    return pp - pages;
}
```

到这里，我们不妨来熟悉一个这个链表的一些基本寻址操作

规定：

> elm是当前元素的指针，类型为Type*。
>
> Type结构体中的链表项（即两个指针组成的结构体）记作field。
>
> 下面代码均假定在宏中编写，因此会有相比起正常代码多余的()

当前元素的前一个元素的le_next指针：

```
*((elm)->field.le_prev)
```

指向当前元素的后一个元素的指针：

```
LIST_NEXT((elm), field)    // 其实就是elm->field.le_next
```

当前元素的后一个元素的le_next指针 / 指向当前元素的下下一个元素的指针：

```
LIST_NEXT(LIST_NEXT((elm), field), field)
```

宏函数LIST_INSERT_TAIL的实现

实现方式1

```
#define LIST_INSERT_TAIL(head, elm, field) do {                     \
            if(LIST_EMPTY(head)){                                   \
                LIST_INSERT_HEAD((head), (elm), field);             \
            }                                                       \
            else{                                                   \
                LIST_FOREACH(LIST_NEXT((elm), field), head, field){ \
                    if(LIST_NEXT(LIST_NEXT((elm), field), field) == NULL){ \
                        break;                                      \
                    }                                               \
                }                                                   \
                LIST_NEXT(LIST_NEXT((elm), field), field) = (elm);  \
                (elm)->field.le_prev = &( LIST_NEXT((elm), field)->field.le_next
);\
                LIST_NEXT((elm), field) = NULL;                     \
            }                                                       \
} while(0)
```

其中 `LIST_FOREACH` 定义如下：

```
#define LIST_FOREACH(var, head, field)                                  \
    for ((var) = LIST_FIRST((head)); (var);  (var) = LIST_NEXT((var), field) )
```

`LIST_FIRST` 定义如下：

```
#define LIST_FIRST(head)        ((head)->lh_first)
```

`LIST_EMPTY` 定义如下：

```
#define LIST_EMPTY(head)        ((head)->lh_first == NULL)
```

上面的实现方式其实有点绕。

最开始 `LIST_NEXT((elm), field)` 也就是elm的le_next指针肯定是null，在for的初始化中令 `LIST_NEXT((elm), field)` 成为 Head 结构体指向链表第一个元素的指针。

然后在for循环体中检查 `LIST_NEXT( LIST_NEXT((elm), field) , field )` 是否是NULL，也就是检查Head后的链表的第一个元素的le_next是不是null（因为第一个if已经检查过链表的第一个元素非空）。如果不是，for中令 `LIST_NEXT((elm), field)` 也就是elm的le_next指针指向链表第二个元素，依次循环下去。直到检查到 `LIST_NEXT( LIST_NEXT((elm), field) , field )` 是NULL，此时，elm的le_next指针指向的自然是链表中的最后一个元素，它的le_next是NULL。

接着，令原来是NULL的 `LIST_NEXT(LIST_NEXT((elm), field), field)` 指向elm，elm的le_prev存储链表最后一个元素的le_next指针的地址，也就是elm的le_next指针的地址 `&( LIST_NEXT((elm), field)->field.le_next )` 。最后再把elm的le_next指针置为NULL。

实现方式2

```
#define LIST_INSERT_TAIL(head, elm, field) do { \
    if (LIST_FIRST((head)) != NULL) { \
        LIST_NEXT((elm), field) = LIST_FIRST((head)); \
        while (LIST_NEXT(LIST_NEXT((elm), field), field) != NULL) { \
            LIST_NEXT((elm), field) = LIST_NEXT(LIST_NEXT((elm), field), field);
\
        } \
        LIST_NEXT(LIST_NEXT((elm), field), field) = (elm); \
        (elm)->field.le_prev = &LIST_NEXT(LIST_NEXT((elm), field), field); \
        LIST_NEXT((elm), field) = NULL; \
    } else { \
        LIST_INSERT_HEAD((head), (elm), field); \
    } \
} while (0)
```

其中 `LIST_FIRST(head)` 宏函数定义如下：

```
#define LIST_FIRST(head)        ((head)->lh_first)
```

这个就相对好理解的多。在判断head对应的链表不为空以后，直接让elm的le_next指针指向链表的第一个元素，相当于第一个元素就是elm此时的下一个元素，然后反复判断elm的下一个元素的下一个元素是不是NULL，如果不是就使得elm的le_next指针由elm的下一个元素指向elm的下一个元素的下一个元素。直到遇到NULL为止。

当循环停止的时候，此时elm的le_next指针指向链表的最后一个元素，因为链表的最后一个元素的下一个元素是NULL，满足elm的下一个元素的下一个元素是NULL。

这时候，让le_next指针指向的（即 `LIST_NEXT((elm), field)` 指向的）链表的最后一个元素的le_next指针指向elm，而elm的le_prev指针指向最后一个元素的le_next指针，而此时这个指针可以由 `LIST_NEXT(LIST_NEXT((elm), field), field)` 表示（elm的下一个元素的le_next指针）。所以加一个取址符。靠着elm的le_next指针找到链表最后一个元素的le_next指针以后，此时就可以把它置为NULL了。

Bonus 如何在上面的链表中访问到elm元素的前一个元素?

通过 `*(elm->field.le_prev)` 只能拿到前一个元素的le_next指针。但是我们通过指针是没办法得到指向前一个元素本身的指针的。那么我们该如何做到这一点呢?

上面的问题其实可以转换为：**已知结构体type的成员member的地址ptr，求解结构体type的起始地址。**

总体的思路便是：type的起始地址 = ptr - size　(这里需要都转换为char *，因为它为单位字节)。

其中size是结构体type的成员member到结构体起始地址的字节数大小。那么怎么求size呢?

这里需要用到一个叫做 `container_of` 的宏。该宏定义如下：

```
#define container_of(ptr, type, member) ({                  \
const typeof( ((type *)0)->member ) *__mptr = (ptr);    \
(type *)( (char *)__mptr - offsetof(type,member) );})
```

这里出现了一个老朋友，在lab1的代码中就已经出现的宏 `offsetof(type, member)`

```
#define offsetof(type, member)  ((size_t)(&((type *)0)->member))
```

这个宏是一个size_t类型的值。

> size_t是标准C库中定义的，在64位系统中为long long unsigned int，非64位系统中为long unsigned int。
>
> size_t是一个基本的**无符号整数**的C / C + +类型，它是sizeof操作符返回的结果类型，该类型的大小可选择。因此，它可以存储在理论上是可能的任何类型的数组的最大大小。换句话说，一个指针可以被安全地放进为size_t类型（一个例外是类的函数指针，但是这是一个特殊的情况下）

宏 `offsetof(type, member)` 的意思是

- `(type *)0` 将内存空间的 0 转换成需要的结构体指针
- `(type *)0)->member` 利用这个结构体指针指向某个成员
- `(&((type *)0)->member)` 取这个成员的地址，这里这个值其实就是结构体type的成员member到结构体起始地址的字节数大小
- `((size_t)(&((type *)0)->member))` 将这个成员的地址转化成 `size_t` 类型

那么显然，凭借宏 `offsetof(tyep, member)` 的就可以求出size的值，再转为 `size_t` 类型。那为什么不用结构体type的成员member的地址ptr直接减去size呢?

这里体现了内核编程的严谨性

第二行额外的中间变赋值

```
const typeof( ((type *)0)->member ) *__mptr = (ptr);
```

它的作用是什么呢? 其实没什么作用，但就形式而言 _mptr = ptr，那为什么要要定义一个一样的变量呢？？？ 其实这正是内核人员的牛逼之处：如果开发者使用时输入的参数有问题：ptr与member类型不匹配，编译时便会有warnning， 但是如果去掉改行，那个就没有了，

最后再说下typeof

这个关键字在宏里面大量使用。常见用法一共有以下几种。

- 不用知道函数返回什么类型，可以使用typeof()定义一个用于接收该函数返回值的变量

▶ 展开查看举例代码

- 在宏定义中动态获取相关结构体成员的类型

```
#define max(x, y) ({                    \
    typeof(x) _max1 = (x);              \
    typeof(y) _max2 = (y);              \
    (void) (&_max1 == &_max2);          \
    _max1 > _max2 ? _max1 : _max2; })
//如果调用者传参时，x和y两者类型不一致，在编译时就会发出警告。
```

因此第二行额外的中间变赋值

```
const typeof( ((type *)0)->member ) *__mptr = (ptr);
```

其实就是利用传入的type参数和member参数，声明一个type结构体的成员member类型的指针承接ptr参数。如果传入的ptr参数类型不是type结构体的成员member类型的指针，那么就会产生warning。在一切正常的情况下，执行第三行代码，通过将ptr-size的结果转为type*类型，得到了成员所在结构体的初始地址。

```
(type *)( (char *)__mptr - offsetof(type,member) );
```

那么回到最初的问题，在通过 `*(elm->field.le_prev)` 拿到前一个元素的le_next指针后，我们自然而然就可以通过宏 `container_of(ptr, type, member)` 用代码访问到前一个元素。

```
container_of(*(elm->field.le_prev), Page, le_next);
```

$Part2.2$ 两级页表相关

细品指导书

MOS 中，采用 `PADDR` 与 `KADDR` 这两个宏就可以对 位于 `kseg0` 的虚拟地址和对应的物理地址进行转换。 它们已经在前面有所提到过。

对于位于 `kuseg` 的虚拟地址，MOS 中采用两级页表结构对其进行管理。

第一级表称为页目录 (Page Directory)，第二级表称为页表 (Page Table)。为避免歧义，下面用 **一级页表** 指代 Page Directory，**二级页表** 指代 Page Table。

两级页表机制相比计组理论中学习的单级页表机制，将虚拟页号进一步分为了两部分。

具体来说，对于一个 32 位的虚存地址，从低到高从 0 开始编号， 其 31-22 位表示的是一级页表项的偏移量，21-12 位表示的是二级页表项的偏移量，11-0 位表示的是页内偏移量。

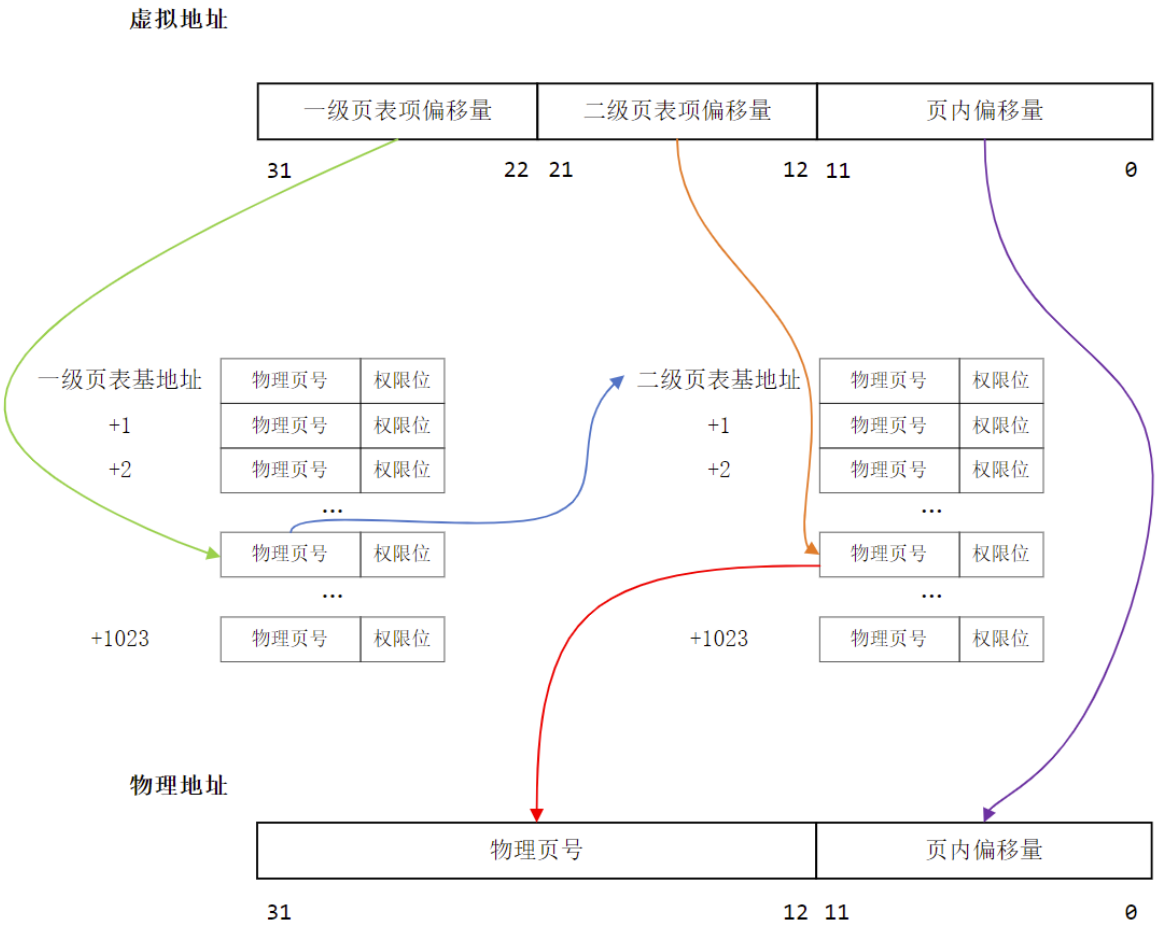
`include/mmu.h` 中提供了两个宏以快速获取偏移量， `PDX(va)` 可以获取虚拟地址 `va` 的 31-22 位， `PTX(va)` 可以获取虚拟地址 `va` 的 21-12 位。

访存时，先通过一级页表基地址和一级页表项的偏移量，找到对应的一级页表项，得到对应的二级页表基地址的物理页号，再根据二级页表项的偏移量找到所需的二级页表项，进而得到对应物理页的物理页号。

这里补充二级页表基地址和其对应的物理页号之间的转化关系

物理页号（一级页表项的高20位）左移12位后（相当于该一级页表项的低12位置零），再加上0x8000 0000得到了二级页表基地址（虚拟地址）。理解---^---：因为物理页号本身对应的物理地址是一定整除页面大小的

图解流程如下：



CPU 发出的地址均为虚拟地址，因此获取相关物理地址后，需要转换为虚拟地址再访问。对页表进行操作时处于内核态，因此使用宏 `KADDR` 即可。

> 存储上，无论是一级页表还是二级页表，它们的结构都是一样的，只不过每个页表项记录的物理页号含义有所不同。每个页表均由 1024 个页表项组成，每个页表项由 32 位组成，包括 20 位物理页号以及 12 位标志位。**关于标志位的相关定义，其实前面已经提到过了，因为每个页表项最后都会填入 TLB 中，因此它的规范和 EntryLo 寄存器的规范相同。** 每个页表所占的空间为 4KB，恰好为一个物理页面的大小。

“每个页表均由 1024 个页表项组成”和“每个页表所占的空间为 4KB”说明每个页表项大小是4字节。因此，指导书才会说：

> 可以发现，一个页表项可以恰好由一个 32 位整型来表示

因此 我们使用 `Pde` 来表示一个一级页表项，用 `Pte` 来表示一个二级页表项，这两者的本质都是 `u_long` ，它们对应的 `typedef` 位于 `include/mmu.h` 。

```
typedef u_long Pde;
typedef u_long Pte;
```

这样，如果是他们类型的指针 `Pde*` `Pte*` ，只需要进行指针的加减，就可以进行以页表项大小为单位的移动。例如，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` 的页表项地址。

现在我们回过头来看一些重要的宏，就会有很清晰的理解。

```
#define BY2PG       4096        // bytes to a page
#define PDMAP       (4*1024*1024)   // bytes mapped by a page directory entry
#define PGSHIFT     12
#define PDSHIFT     22      // log2(PDMAP)
#define PDX(va)     ((((u_long)(va))>>22) & 0x03FF)
#define PTX(va)     ((((u_long)(va))>>12) & 0x03FF)
#define PTE_ADDR(pte)   ((u_long)(pte)&~0xFFF)  // 将低12位置零
```

`BY2PG` 代表一页的大小，

`PDMAP` 代表一个**由一个一级页表所映射的1024个二级页表的大小**，具体信息诸如页表项大小在前面已经说了。

`PTE_ADDR` 将地址的低12位置零。则相当于将二级页表pte的页表项的12位权限位 置零

页表项的具体内容相关宏

```
#define PTE_G      0x0100 // Global bit
#define PTE_V      0x0200 // Valid bit   从左往右第10位（2^9）
#define PTE_R      0x0400 // "Dirty", but really a write-enable bit. 1 to allow
writes, 0 and
            // any store using this translation will be trapped.
#define PTE_D      0x0002 // fileSystem Cached is dirty
#define PTE_COW        0x0001 // Copy On Write
#define PTE_UC     0x0800 // unCached
#define PTE_LIBRARY        0x0004 // share memmory
```

PTE_V这一位是1时，代表这个页表项是有效的。valid bit有效位

相关函数详细解析

首先提一嘴， `E_NO_MEM` 是定义在mmu.h的宏，就是一个数字4

boot_pgdir_walk()

它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（10位一级页表偏移量+10位二级页表偏移量+12位页内偏移量）所表示的二级页表项的地址，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。这里之所以使用 alloc 而不用 page_alloc 是因为这个函数是在内核启动过程中使用的，此时还没有建立好物理内存管理机制。

```
static Pte *boot_pgdir_walk(Pde *pgdir, u_long va, int create)
{
    Pde *pgdir_entry;
    Pte *pgtable, *pgtable_entry;
    /* Step 1: Get the corresponding page directory entry and page table. */
    pgdir_entry = pgdir + PDX(va);  // 基地址+一级页表偏移地址(va的22~31位) 得到二级页
表基地址对应的物理页号所存放在 的页表项地址，存入pgdir_entryp
    // 对于pages的第一次调用来说，pgdir是0x80401000，va是UPAGES+0=0x7f800000，取22~31
位是0x1fd
    // 则pgdir_entryp = 0x804011fd
    // 事实上，由于内存控制块总共就申请0x3 0000 空间，每次这里都是取22~31位，所以一级页表偏
移量不变，pgdir_entryp不变。因此，整个申请过程中，始终是在同一个一级页表项里操作。
    pgtable = KADDR(PTE_ADDR(*pgdir_entry));   // 将这个页表项存放的内容（二级页表基地
址对应的物理页号+权限位）的低12位（权限位）置零后，转换为虚拟地址（就是最高位置1，加上0x8000
0000），存入pgtable。此时，这个值就是二级页表的基地址
    /* Step 2: If the corresponding page table is not exist and parameter
`create`
     * is set, create one. And set the correct permission bits for this new page
     * table. */
    // 如果一级页表的页表项内容中的有效位是0 --- 在这里的含义是对应的二级页表不存在
    // 这里因为mips_vm_init为一级页表申请空间以后，这页空间全是0，所以有效位肯定是0
    // 所以在create=1的情况下，需要申请一页二级页表
    if ((*pgdir_entry & PTE_V) == 0) {
        if (create) {
            pgtable = (Pte*) alloc(BY2PG, BY2PG, 1);
            *pgdir_entry = PADDR(pgtable) | PTE_V; // | PTE_V 将PTE_V代表的那一位置
1，表示页表项有效，对应的二级页表 现在存在了 这里不设置可读位是因为这里启动的映射是只读的，但是
设置也可以
        } else return 0; // exception
    }
    /* Step 3: Get the page table entry for `va`, and return it. */
```

```
    pgtable_entry = &pgtable[PTX(va)]; // 二级页表的基地址patable + 二级页表的偏移地址
(va的12~21位) 得到va 这个虚拟地址所在的二级页表项，再用&取它的地址
    // 第i次调用时，va是UPAGES + i*BY2PG = 0x7f80i000，总共调用内存控制块占用的页数:
3*2^16 / 2^12 = 3*2^4 = 48页
    // 则PTX(va)就是[0, 47]的48个整数，这又一次说明了只需要一个二级页表就可以
    return pgtable_entry;
}
```

boot_map_segment()

它的作用是将一级页表基地址 pgdir 对应的两级页表结构做区间地址映射，将虚拟地址区间 [va, va + size – 1] 映射到物理地址区间 [pa, pa + size – 1]，因为是按页映射，要求 size 必须是页面大小的整数倍。同时为相关页表项的权限为设置为 perm。这个函数调用了 boot_pgdir_walk，它也是在内核启动过程中使用的，前面我们提到过它为 Page 结构体和 Env 结构体进行了映射。

```
void boot_map_segment(Pde *pgdir, u_long va, u_long size, u_long pa, int perm)
{
    int i, va_temp;
    Pte *pgtable_entry;
    /* Step 1: Check if `size` is a multiple of BY2PG. */
    if (size % BY2PG){
        return;
    }
    /* Step 2: Map virtual address space to physical address. */
    /* Hint: Use `boot_pgdir_walk` to get the page table entry of virtual
address `va`. */
    // 物理内存控制块需要循环48次，进程控制块需要循环54次
    for (i = 0, size = ROUND(size, BY2PG); i < size ; i+=BY2PG) {
        va_temp = va + i;
        /* I. 找到va_temp表示的二级页表项地址，存入pgtable_entry这个指针 */
        pgtable_entry = boot_pgdir_walk(pgdir, va_temp, 1);
        /* II. 用pgtable_entry这个指针在二级页表项的内容中存放物理地址并设置权限 */
        // 不难想到，这里用到的传入的物理地址pa一定是物理页号对应的物理地址，也就是说低12位必
然全是0（整除页面大小），否则像下面这样这样直接进行 或运算，可能会丢失信息
        *pgtable_entry = (pa + i) | perm | PTE_V;
    }
    pgdir[PDX(va)] |= (perm | PTE_V); // ???
}
```

pgdir_walk()

它返回一级页表基地址 pgdir 对应的两级页表结构中，va 这个虚拟地址所对应的二级页表项的地址。这个函数是 boot_pgdir_walk 的非启动版本，只不过后者是在内核启动过程，不允许失败，而该函数是在普通运行过程，如果空间不够允许失败，因此将返回值变为 int，若为 0 代表执行成功否则为一个失败码，同时**将原本的返回值 Pte * 放到 ppte 所指的空间上**。此外，因为该函数的调用在启动之后，create 不为 0 且对应的二级页表不存在时，它使用 page_alloc 而不使用 alloc 进行物理页面的分配。

```
int pgdir_walk(Pde *pgdir, u_long va, int create, Pte **ppte)
{
    Pde *pgdir_entryp;
    Pte *pgtable;
    struct Page *ppage;
    int ret;
    /* Step 1: Get the corresponding page directory entry and page table. */
    pgdir_entryp = pgdir + PDX(va);  // 基地址+一级页表偏移地址(va的22~31位) 得到二级
页表基地址对应的物理页号所存放在 的页表项地址，存入pgdir_entryp
```

```
    /* Step 2: If the corresponding page table is not exist(valid) and parameter
`create`
     * is set, create one. And set the correct permission bits for this new page
table.
     * When creating new page table, maybe out of memory. */
    // 如果一级页表的页表项内容中的有效位是0 --- 在这里的含义是对应的二级页表不存在
    if ((*pgdir_entryp & PTE_V) == 0) {
        if (create) {
            if ((ret = page_alloc(&ppage)) < 0) return ret;
            ppage->pp_ref++; // 注意！新创建的页 有映射关系，所以ref要加1
            // 这里操作的是为一级页表申请的二级页表，这一页虚拟对应的物理页的引用要加1
            *pgdir_entryp = page2pa(ppage)
                            /* Physical Address of `page` */
                          | PTE_V | PTE_R;
        } else {
            *ppte = 0;
            return 0;
        }
    }
    /* Step 3: Set the page table entry to `*ppte` as return value. */
    pgtable = (Pte *)KADDR(PTE_ADDR(*pgdir_entryp));  //依然是用 PTE_ADDR这个函数将
页表项内容的低12位清零。再转为虚拟地址。得到二级页表的基地址（虚拟地址）
    *ppte = pgtable + PTX(va); // 得到具体需要的二级页表项的地址，和上面boot_page_walk
函数中的数组取址方式效果是一样的。
    return 0;
}
```

page_insert()

> 这个函数的作用是将一级页表基地址 pgdir 对应的两级页表结构中 va 这一虚拟地址映射到内存控制块 pp 对应的物理页面，并将页表项权限为设置为 perm。

其实，用更加易懂的话来说，应该是，将一级页表基地址 pgdir 对应的两级页表结构中 va 这一虚拟地址**所对应的二级页表项中填入地址 —— pp这个物理内存控制块对应的物理页面的地址，并设置页表权限。**

所有这种建立映射关系的函数，他们的参数 va 仅仅是他们的参数 页目录 所映射的4G大小中的va对应的位置，而不是内核地址的4G中的位置。

注意，`int pgdir_walk(Pde *pgdir, u_long va, int create, Pte **ppte)` 将原本的返回值 Pte * （va 这个虚拟地址所在的二级页表项的地址）放到 ppte 所指的空间上 `*ppte = pgtable + PTX(va)` 。

此外，读到这里，依然需要牢记一点：函数中操作的始终是内存控制块的地址而不是真正的物理页面的地址，只是通过内存控制块和物理页面的对应关系，构建一种逻辑上的映射关系。具体复习前面的内容内存控制块相关。

```
int page_insert(Pde *pgdir, struct Page *pp, u_long va, u_int perm)
{
    u_int PERM;
    Pte *pgtable_entry;
    PERM = perm | PTE_V;
    /* Step 1: Get corresponding page table entry. */
    // I. check whether `va` is already mapping to `pa`
    pgdir_walk(pgdir, va, 0, &pgtable_entry); // pgtable_entry 本身被设为了 二级页
表项的地址，此时它就相当于指向va对应的二级页表项的指针。
    // pgtable_entry的值不为0（结合pgdir_walk函数在 create=0 时的输出，这说明二级页表存
在） 且 pgtable_entry指向的内容（二级页表项存放的内容，即物理页号+权限位）中权限位的有效位为1
--- 说明va对应的二级页表项已经存在了 对物理页面的映射
    if (pgtable_entry != 0 && (*pgtable_entry & PTE_V) != 0) {
        // II. check whether `va` is mapping to another physical frame
        // 将物理地址转为物理页数，如果物理页数 对应的控制块的地址不是pp  取消这个映射
```

```
        if (pa2page(*pgtable_entry) != pp) {
            page_remove(pgdir, va);       // unmap it!
        } else  { // 如果就是pp，那么已经不需要建立映射了，只需要更新tlb 和 更新pp对应物理页面基地址的权限（因为在lab4里面会给父进程的页表项新增PTE_COW用于保护，lab5里面会新增PTE_LIBRARY）
            tlb_invalidate(pgdir, va);              // <~~
            *pgtable_entry = (page2pa(pp) | PERM);  // update the permission
            return 0;
        }
    }
    // 如果不符合上面判断条件 --- 说明 va对应的二级页表项  并不存在对物理页面的映射
    /* Step 2: Update TLB. */
    tlb_invalidate(pgdir, va); // 使得tlb中 va 对应的这一项无效（清零tlb表项）

    /* Step 3: Do check, re-get page table entry to validate the insertion. */

    /* Step 3.1 Check if the page can be insert, if can't return -E_NO_MEM */
    // 不符合上面的判断语句，有可能是对应的二级页表根本不存在，此时pgtable的值被赋值为0（因为上面的create是0），那么这里就把create传入1，再试一次，如果没有二级页表就直接创建一个，出意外就报内存错误。此时新创建的页面已经在pgdir_walk中更新了权限位的 有效位 和 可写位
    if(pgdir_walk(pgdir, va, 1, &pgtable_entry)){
        return -E_NO_MEM;
    }
    /* Step 3.2 Insert page and increment the pp_ref */
    *pgtable_entry = page2pa(pp) | PERM; // 将这个二级页表项的内容 和 pp内存控制块对应的物理页表构建映射，并设置二级页表项的权限。
    pp->pp_ref++;
    return 0;
}
```

关于 `(pgtable_entry != 0 && (*pgtable_entry & PTE_V) != 0)` 这个判断条件更加清晰的解释可以参见 `page_lookup()` 函数。

两次ref

准确的来说，pagewalk建立的是一级到二级的引用，引用的是二级页表对应的那一页物理内存的引用次数，pageinsert是二级到物理的引用，是二级页表中存放的物理地址对应的物理页的引用次数。

page_decref()

```
// Overview:
//  Decrease the `pp_ref` value of Page `*pp`, if `pp_ref` reaches to 0, free this page.
void page_decref(struct Page *pp) {
    if(--pp->pp_ref == 0) {
        page_free(pp);
    }
}
```

page_lookup()

找到并返回虚拟地址 va 映射到的 **物理内存控制块** 的地址 的函数，并将 va 对应的二级页表项的指针存入Pte **ppte（二级指针，*ppte 即指向二级页表项的指针Pte *pte），当然，当不需要拿到这个二级页表项的指针时，将这个参数置为NULL即可。（函数中会直接跳过存放）

```
/*Overview:
  Look up the Page that virtual address `va` map to.

  Post-Condition:
```

```
  Return a pointer to corresponding Page, and store it's page table entry to
*ppte.
  If `va` doesn't mapped to any Page, return NULL.*/
struct Page *page_lookup(Pde *pgdir, u_long va, Pte **ppte)
{
    struct Page *ppage;
    Pte *pte;

    /* Step 1: Get the page table entry. */
    pgdir_walk(pgdir, va, 0, &pte);  // 首先找到va在pgdir的二级页表结构中对应的二级页表
项地址
    /* Hint: Check if the page table entry doesn't exist or is not valid. */
    if (pte == 0) { // pte为0  结合create为0时pgdir_walk的返回值，说明二级页表不存在
        return 0;
    }
    if ((*pte & PTE_V) == 0) { // va在pgdir的二级页表结构中对应的二级页表项 是无效的，说
明二级页表存在，但是va对应的二级页表项没有记录物理内存控制块，即va所在的一页虚拟内存 不存在 和
物理内存的映射
        return 0;    //the page is not in memory.
    }

    /* Step 2: Get the corresponding Page struct. */
    ppage = pa2page(*pte);
    if (ppte) {       // 将va对应的二级页表项的地址存入参数 ppte
        *ppte = pte;
    }
    return ppage;
}
```

page_remove()

page_remove单纯就是一个 解除**物理内存控制块**和**页表中的虚拟地址**的**映射关系** 的函数。

具体来说，就是将pgdir页目录映射的二级页表体系下的虚拟地址va所对应的二级页表项中，存放的物理页号对应的物理页面的引用次数减一，如果引用次数为零就调用 `page_free` 函数。这一点是通过 `page_lookup` 函数找到该物理页面的物理内存控制块实现的。

然后，将pgdir页目录映射的二级页表体系下的虚拟地址va所对应的二级页表项的内容清零。

最后更新TLB的表项。

```
// Overview: Unmaps the physical page at virtual address `va`.
void page_remove(Pde *pgdir, u_long va)
{
    Pte *pagetable_entry;
    struct Page *ppage;
    /* Step 1: Get the page table entry, and check if the page table entry is
valid. */
    ppage = page_lookup(pgdir, va, &pagetable_entry);
    // page_lookup函数返回值是va对应的物理内存控制块的地址，同时能将参数中的
pagetable_entry设置为二级页表项的指针
    if (ppage == 0) {
        // 结合 page_lookup 返回0的情形 说明二级页表不存在/二级页表项无效（不在内存中）
        // 也就是说虚拟地址 va 并没有对应的物理页面
        return;
    }
    /* Step 2: Decrease `pp_ref` and decide if it's necessary to free this page.
*/
    /* Hint: When there's no virtual address mapped to this page, release it. */
    ppage->pp_ref--;
    if (ppage->pp_ref == 0) {
        page_free(ppage);
    }
```

```
    /* Step 3: Update TLB. */
    *pagetable_entry = 0;     // 利用二级页表项的指针，使得二级页表项内容清零。
    // 即：取消  虚拟地址va  到  原来va对应的二级页表项中存储的物理地址  的映射
    tlb_invalidate(pgdir, va); // 使得tlb中 va 对应的这一项无效（清零tlb表项）
    return;
}
```

Part2.3 TLB

这里再复习一下 TLB 的功能。MOS 中，通过页表进行地址变换时，硬件只会查询 TLB，如果查找失败，就会触发 TLB 中断，对应的异常中断处理就会对 TLB 进行重填。需要特别注意的是，lab2 里面**没有开启** 这个功能，因此上述过程你无法测试出来，但你一定要知道它的原理。这一部分的详细内容会在下一部分进行介绍。

使用 `tlb_invalidate` 函数可以实现删除特定虚拟地址的映射，**每当页表被修改，就需要调用该函数以保证下次访问该虚拟地址时诱发 TLB 重填以保证访存的正确性。**

除此之外，一般的操作系统中，当物理页框全部被映射，此时有新的虚拟页框需要映射到物理页框，那么就需要将一些物理页框 置换到硬盘中，选择哪个物理页框的算法就称为页面置换算法，例如我们熟悉的 FIFO 算法和 LRU 算法。

然而在我们的 MOS 中，对这一过程进行了简化，一旦物理页框全部被分配，进行新的映射时并不会进行任何的页面置换，而是直接返回错误，这对应 `page_alloc` 函数中返回的 `-E_NO_MEM` 。

TLB 的结构

（基于强生大佬的博客）

首先我们需要弄懂 tlb 的结构，计组认为的 TLB，是长这样的：

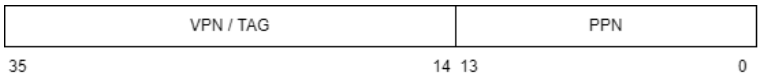
也就是说，TLB 是一个全相连的 cache，既然是全相连，就不需要 index 段了。我们用虚拟地址的前 22 位作为 TAG，并行的比较 64 个TLB 的 line，如果 TAG 相等，就说明找到了，反之，这说明没有找到。

不过这个模型还是有些粗糙的，很多细节并没有说明白。

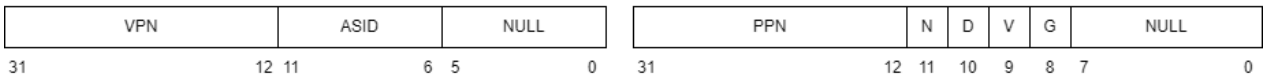
在操作系统指导书里提到，tlb 构建了一个映射关系，简化一下就是 $VPN \rightarrow PPN$。当然这是对的了，但是这种说法似乎是说每个 VPN 都会对应一个 PPN，但是其实这种映射关系只有 64 对。而且叫映射似乎就是一下就射过去了，而不是一个并行的比较过程

其次就是，我们没有了解具体硬件发生了啥，比如 VPN 是怎样被检索的，被检索到的 PPN 放到了哪里，tlb 缺失以后具体怎么填补。都是没有的。这其实跟协处理器有很大关系。

在了解协处理器之前，我们先来看一下 tlb 的表项，他比计组版本要复杂一些，我们以 MOS 中 64MB（也就是共有 2^{14} 页 ）的物理内存为例。



朴素版



真实版

我们来说明一下这些差别：

- 朴素版的 PPN 只有 14 位，是因为物理页框号可以最少用 14 位表示（上面说了MOS系统的物理内存是2^{14}页）。但是真实版的 PPN 也与 VPN 相同，是 22 位。可能是考虑到不同电脑上内存不同吧，这估计也是 mips_detect_memory() 这个函数的设置。
- 朴素版一个TLB的页表项 entry 是 36 位，而真实版一个 entry 是 64 位。这是因为真实版的标志位更多，所以需要的位数就更多
- 朴素版没有 ASID 段，而真实版有。ASID（address space identifier）应该是用于区分不同进程的一个标识符，因为操作系统可以同时运行多个进程，要是不用 ASID 的话，只要进程一切换，那么 TLB 里的所有内容都需要失效（因为进程切换就以为着虚拟地址和物理地址的映射关系切换），而这样是低效的，因为每次 TLB 中的内容清空，就意味着会发生 64 次的冷缺失。
- 朴素版没有物理地址权限标志位（N，D，V，G），而真实版有。这四个标志位的解释见下表

标志位	解释
N（Non-cachable）	当该位置高时，后续的物理地址访存将不通过 cache
D（Dirty）	事实上是可写位。当该位置高时，该地址可写；否则任何写操作都将引发 TLB 异常。
V（Valid）	如果该位为低，则任何访问该地址的操作都将引发 TLB 异常。
G（Global）	如果该位置高，那么允许不同的虚拟地址映射到相同的物理地址，可能类似于进程级别的共享

总结起来就是真实版的 tlb 建立了一个这样的映射 $\langle VPN, ASID\rangle \rightarrow \langle PPN, N, D, V, G\rangle$

每一个 TLB 表项都有 64 位，其中高 32 位是 Key，低 32 位是 Data。TLB总共有64项，因此TLB也是4K大小。

Key（对应到EntryHi）

- VPN: Virtual Page Number
  - 当 **TLB** 缺失（CPU 发出虚拟地址，欲在 TLB 中查找物理地址但未查到）时，**EntryHi** 中的 **VPN** 自动（由硬件）填充为对应虚拟地址的虚页号。
  - 当需要填充或检索 TLB 表项时，软件需要将 VPN 段填充为对应的虚拟地址
- ASID：Address Space IDentifier
  - 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）。

Data（对应到EntryLo）

- PFN：Physical Frame Number

    - 软件通过填写 **PFN**，随后使用 **TLB** 写指令，才将此时的 **Key** 与 **Data**写入 **TLB** 中。
- N、D、V、G在上面的表格中以及说了。

TLB 怎么用

然后我们来解决下一个问题，就是 tlb 怎么用的问题。这是一个我之前忽略的点，因为其实我对于 tlb 的定位并不清楚，我本以为它就好像是一个 cache，是对于程序员是透明的，我就在编程的时候写虚拟地址，然后就有硬件（MMU）拿着这个地址去问 tlb，tb再做出相关反应，这一切都是我不需要了解的，但是实际上 tlb 的各种操作，都是需要软件协作的。之所以有这个错误认知，是因为似乎在 X86 架构下确实是由硬件干的，但是由于我们的 MIPS 架构，也就是 RISC 架构，所以似乎交由软件负责效率更高一些。

如果 tlb 是程序员可见的，那么我们必然要管理它，那么我们就需要思考怎样管理它？我们管理它的方式就是设置了专门的寄存器和专门的指令。指令用于读或者写 tlb 中的内容，而寄存器则用于作为 CPU 和 tlb 之间沟通的媒介，就好像我们需要用 hi 和 lo 寄存器与乘除单元沟通一样。这些寄存器，都位于 CP0 中。

在协处理器里面与 tlb 有关的寄存器如下表：

寄存器	编号	作用
EntryHi	10	保存某个 tlb 表项的高 32 位，任何对 tlb 的读写，都需要通过 EntryHi 和 EntryLo
EntryLo	2	保存某个 tlb 表项的低 32 位
Index	0	决定索引号为某个值的 tlb 表项被读或者写
Random	1	提供一个随机的索引号用于 tlb 的读写

这里再说一下各个寄存器的域

- EntryHi，EntryLo 的域分别与 tlb 表项的 高32位 和 低32位 完全相同
- Index的域如下图所示。**这里注意到，index域中的Index(8~13位)恰好是6位，$2^6 = 64$项，刚好是TLB的表项数目。因此这里index域的Index部分就是TLB的表项索引。**

Figure 6.3. Fields in the Index register

- Random域如下图所示

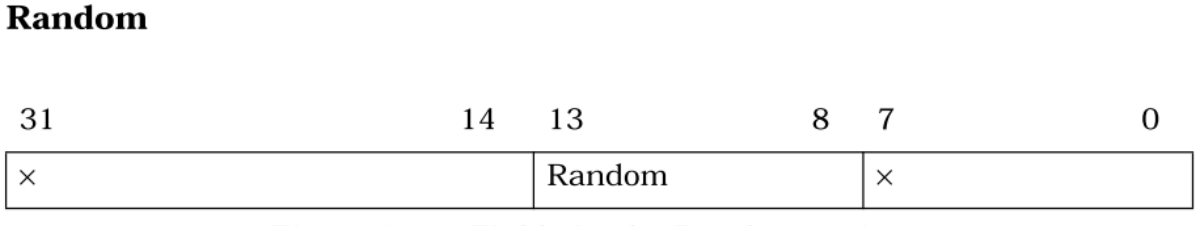


Figure 6.4. Fields in the Random register

与 tlb 相关的指令

指令	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。
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	tlb probe。用于查看 tlb 是否可以转换虚拟地址（即命中与否）。根据 EntryHi 中的 Key（包含 VPN 与 ASID），查找 TLB 中与之对应的表项。如果命中，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。

然后我们可以结合前面的tlb_out()函数的解析理解上面的指令。

$Part2.4$ 多级页表和页表目录自映射部分

细品指导书

在 Lab2 中，我们实现了内存管理，建立了两级页表机制。

试想这样一个问题：如果页表和页目录没有被映射到进程的地址空间，而一个进程的 4GB 地址空间均映射物理内存的话，那么就需要 4MB 来存放页表（需要4GB/(4KB*1024)=1024 个页表，每个页表大小是4KB：1024个4字节页表项（32位）），4KB 来存放页目录（1024个页表需要一个页目录表示）；

而在 MOS 中，页表和页目录都在进程的地址空间中得到映射，这意味着在 1024 个页表中，**有一个页表所映射的4MB空间就是这1024个页表实际占用的4MB空间**（一个页表是1024项，一项对应一页物理内存4KB，因此一个页表对应4MB物理内存）。这一个特殊的页表就是页目录，它的 1024 个表项映射到这 1024 个页表。因此只需要 4MB 的空间即可容纳页表和页目录。

> 而 MOS 中，将页表和页目录映射到了用户空间中的 0x7fc00000-0x80000000（共4MB）区域，这意味着 MOS 中允许在用户态下访问当前进程的页表和页目录，这一特性将在后续的 Lab 中用到。

这里的位置是UVPT

```
#define ULIM 0x80000000
#define UVPT (ULIM - PDMAP)
#define UPAGES (UVPT - PDMAP)
#define UENVS (UPAGES - PDMAP)
```

那么到这里，结合mips_vm_init()函数，ULIM（0x8000 0000）下面的三个PDMAP存放的东西就彻底明白了：

UVPT - ULIM的4M空间里，映射的是页表和页目录

UPAGES - ULIM的4M空间里，映射的是Pages结构体数组pages，即物理内存页面控制块

UENVS - UPAGES的4M空间里，映射的是Env结构体数组envs，即进程管理控制块

二级页表目录自映射

二级页表目录自映射的总体逻辑是：首先先用4M的二级页表区映射4G的虚拟空间，然后再用二级页表中的某一个二级页表（4K）作为一级页表表示这些二级页表（4M）。

首先是虚拟空间（4G）到二级页表区（4M）的映射

首先，用含有 1M 个页表项的二级页表区（1024个页表）映射4G的虚拟空间（4K为基本单位，共2^20页），则一个页表项代表着4G/1024=4M=2^12 B大小的虚拟空间（恰好是1页），则二级页表区起始地址0x7fc0 0000对应的是二级页表区中的第 (0x7fc0 0000)>>12 个页表项，而每个页表项大小是4B，则二级页表区起始地址0x7fc0 0000在二级页表区内的偏移量是((0x7fc0 0000)>>12)<<2 = 0x1ff000 B，即在二级页表区中对应的地址是0x7fc0 0000 + 0x1ff000 = 0x7fdff000 B。

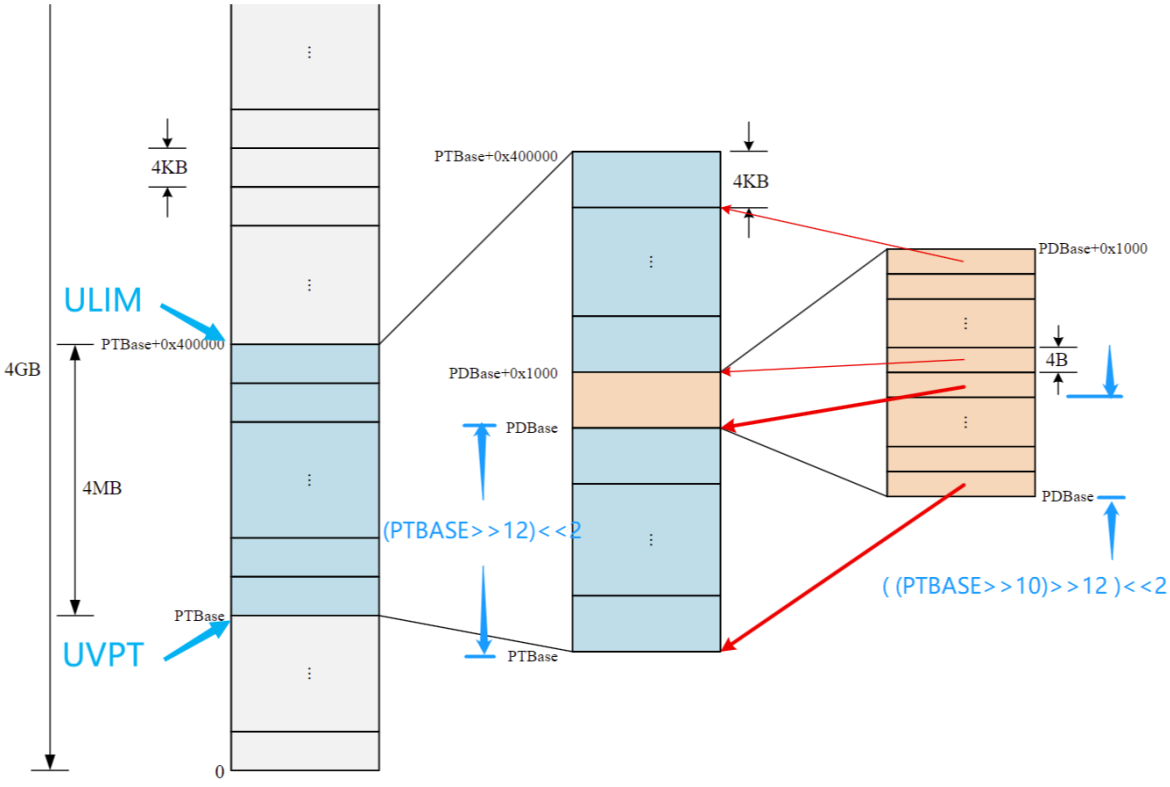
> 我的猜想：因此这里最底层的4G虚拟内存，可以看作是最低级的页表，因为二级页表区的每一个页表项都管理虚拟内存的一页。

然后是二级页表区（4M）到一级页表/页目录（4K）的映射

然后，再用二级页表区起始地址0x7fc0 0000在二级页表区中对应的地址0x7fdff000 的这一个页表作为一级页表，映射整个二级页表区。那么0x7fdff000改称为一级页表/页目录基地址。

一级页表在二级页表区中的偏移量是(0x7fc0 0000)>>10，则一级页表对应的地址在一级页表中的偏移量是这个量再右移12位（对应的页表项）乘以4（每个页表项的大小），所以就是(0x7fc0 0000)>>20。

图示



Part3 体会与感想&指导书反馈

这次新的东西的代码量很大，要想学的透彻需要花很长时间。

这次的思考题多多少少有点过分了，说实话，这次的思考题我不知道是不是和实验关系紧密，但是要花很多时间是真的。尤其是写到现在，我写出了2w字的总结，却一度无法写出思考题。

思考题==要看额外的英文资料？可是额外的资料也没在课程资源里给出来，而且并不好找。我是在叶助教的博客里找到了链接。所幸的是，并不难，答案都可以在第六章第一页找到，而且了解了若干知识。但是时间啊……

思考题==要看甚至不在本地分支的代码？可是这种耗时未免太大了，真的承受不起。

思考题==软院同学要从0开始了解x86或者RISC-V的内存管理机制？我是勉强找到了一篇写的比较好的文章，然后和计院同学交流以后勉强模糊地了解了一些。

整个实验中，有那么一两个类似于这样要额外花费大量时间的思考题，我觉得就已经差不多了。这次扎堆出现，我不清楚是怎么想的。虽然确实能收获到东西，但是拜托了助教真的心疼心疼软院的同学吧，真的顶不住了。

我能写出两万字的总结，指导书里显然是缺少必要的引导的。一些代码的解释少之又少，根本不足以串联起来整个流程。我觉得助教高估了同学们的水平，真的。我不是想说助教不用心，而是说，可能助教作为已经通透掌握了这门课的知识的人，在某些细节的理解方面并不会意识到这对于初次接触的同学来说是很有难度的。而且代码真的讲的太少了……这次有同学甚至写不出虚拟页号，难道这还不能反映出什么吗？