OS lab2

Thinking

> **Thinking 2.1** 请根据上述说明，回答问题：在编写的 C 程序中，指针变量中存储的地址被视为虚拟地址，还是物理地址？MIPS 汇编程序中 `lw` 和 `sw` 指令使用的地址被视为虚拟地址，还是物理地址？ ■

2.1

在 C 语言程序中，指针变量的地址是虚拟地址；MIPS 汇编程序中 lw 和 sw 使用的都是虚拟地址（我们可以认为，用户代码中所操作的地址都是虚拟地址，操作系统通过虚实地址映射让用户**以为**自己在操作一段连续的地址空间）

> **Thinking 2.2** 请思考下述两个问题：
>
> - 从可重用性的角度，阐述用宏来实现链表的好处。
> - 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。
>
> ■

2.2

宏定义实现链表的好处：

使用宏定义实现链表的核心优势在于 **代码复用与类型无关性**，具体体现在以下方面：

1. 通用性：宏实现的链表不依赖具体数据类型，只需通过宏参数指定节点结构体的成员名称即可；例如，`LIST_ENTRY` 宏可以用于任何结构体，无需为每种数据类型重写链表逻辑
2. 避免重复代码：传统链表需要为每种数据类型单独实现插入、删除等操作（如 `insert_student()`、`insert_teacher()`），而宏通过抽象操作（如 `LIST_INSERT_HEAD`），只需一套代码即可支持所有类型
3. 编译时展开，无运行时开销：宏在预处理阶段展开为直接操作指针的代码，性能与手写链表相同，且在编译阶段就可以被展开，减少了函数调用的栈开销，能够提高部分性能
4. 灵活的扩展性：宏可轻松支持多种链表变体（如单向、双向、循环链表），只需修改宏定义，无需调整调用代码；例如，`sys/queue.h` 提供了 `LIST`（单向）、`TAILQ`（双向）、`CIRCLEQ`（循环双向）等宏

分析单向链表、循环链表、双向链表之间的差异：

操作	单向链表（LIST）	循环链表（CIRCLEQ）	双向链表（TAILQ）
头插	O(1)	O(1)	O(1)
尾插	O(n)	O(1)	O(1)
中间插入	O(n)	O(n)	O(1)
删除头节点	O(1)	O(1)	O(1)
删除尾节点	O(n)	O(1)	O(1)
删除中间节点	O(n)	O(1)	O(1)

性能对比总结：

1. 宏实现链表在可重用性、类型通用性和性能上具有显著优势
2. 双向链表（TAILQ）在插入/删除操作上性能最优，适合频繁修改的场景；单向链表节省内存但尾插效率低；循环链表适合环形缓冲等特殊需求
3. 实际开发中应优先选择 `sys/queue.h` 的宏实现，而非手写链表

> 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。

2.3

正确的 Page_list 结构的展开：

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

这是一个类似 BSD 风格的链表结构实现，以下是其具体组成分析：

外层结构：Page_list

1. 内部成员 lh_first：指向链表第一个节点的指针，类型是一个匿名结构体的指针，即链表头节点可以通过 lh_first 来访问，初始化为 NULL 则表示空链表
2. 内部成员匿名嵌套结构体：
    1. 成员 pp_link：双向链表的核心链接结构，包括两个成员：
       le_next：指向下一个 Page 节点的指针
       le_prev：指向前一个 Page 节点的 le_next 指针的指针（一个二级指针，这里主要是为了支持 O(1) 的快速节点删除）
    2. 成员 pp_ref：记录每个 Page 物理页的引用计数，也就是被虚拟地址映射的次数
3. 链表节点 Page：链表的实际节点类型为 Page，内部包含 pp_link 成员

> **Thinking 2.4** 请思考下面两个问题：
>
> - 请阅读上面有关 TLB 的描述，从虚拟内存和多进程操作系统的实现角度，阐述 ASID 的必要性。
> - 请阅读 MIPS 4Kc 文档《MIPS32® 4K™ Processor Core Family Software User's Manual》的 Section 3.3.1 与 Section 3.4，结合 ASID 段的位数，说明 4Kc 中可容纳不同的地址空间的最大数量。
>
> ■

2.4

AISD：地址标识符

由于在多线程系统中，对于不同进程，相同的虚拟地址各自映射到不同的物理地址空间，而所有的进程共用同一个 TLB 表项，因此 TLB 中装着不同进程的页表项，ASID用于区别不同进程的页表项，有时候我们修改一个虚拟地址 va 所对应的物理页，它可能在许多进程中都进行了映射，且映射到不同的物理地址，此时我们就需要根据 ASID 的标志找到我们要修改的那一组映射关系

地址空间最大数量：

ASID 6位，容纳64个不同进程

> **Thinking 2.5** 请回答下述三个问题：
>
> - `tlb_invalidate` 和 `tlb_out` 的调用关系？
> - 请用一句话概括 `tlb_invalidate` 的作用。
> - 逐行解释 `tlb_out` 中的汇编代码。
>
> ■

2.5

tlb_invalidate 用来使得某个 ASID 标识符下的虚拟地址 va 和它映射的物理地址失效，用于页表被修改的时候刷新对应的 TLB 表项

它会调用 tlb_out，后者是前者具体的实现形式，我们把要失效的 va 传入 tlb_out 中，即可进行查询，进而使得它对应的页表的 entryHi 和 entryLo 中的键值被清空

代码解析见**下文**

> **Thinking 2.6** 请结合 Lab2 开始的 CPU 访存流程与下图中的 Lab2 用户函数部分，尝试将函数调用与 CPU 访存流程对应起来，思考函数调用与 CPU 访存流程的关系。 ■

2.6

函数调用与 CPU 访存流程的对应关系（从指令执行，栈操作，内存访问三个角度来分析）：

1. 函数调用前的参数传递时 CPU 动作：调用者（如 main 函数）将参数压入栈或存入寄存器（取决于调用约定）；访存操作：通过 push 或 mov 指令将参数写入栈内存（Store 操作）
2. 函数调用时的返回地址保存 CPU 动作：执行 call 指令时，CPU 自动将返回地址（下一条指令的 eip）压栈；访存操作：隐式的 push eip（Store 操作），修改栈指针 esp
3. 函数内部的局部变量访问时 CPU 动作：被调用函数（如 foo）通过栈指针 ebp 访问参数和局部变量，访存操作：

    1. 加载参数：mov eax, [ebp+8]（Load 操作）
    2. 存储局部变量：mov [ebp-4], eax（Store 操作）
4. 函数返回时的栈帧恢复时 CPU 动作：执行 ret 指令时，CPU 从栈中弹出返回地址到 eip，访存操作：隐式的 pop eip（Load 操作）

函数调用对 CPU 访存流程的影响：

栈内存的频繁访问：

1. 函数调用通过栈传递参数、保存返回地址和局部变量，导致大量的 Load/Store 操作；栈指针（esp/ebp）的修改是访存的核心
2. 访存局部性：栈操作具有空间局部性，CPU 缓存（Cache）能有效加速栈访问
3. 流水线冲突风险：频繁的栈内存访问可能导致数据冲突（如 push 和 pop 依赖 esp），需流水线停顿或乱序执行优化

> **Thinking 2.7** 从下述三个问题中任选其一回答：
>
> - 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。
> - 简单了解并叙述 LoongArch 中的内存管理机制，比较 LoongArch 与 MIPS 在内存管理上的区别。
>
> ■

2.7

X86 体系结构中的内存管理机制

1. 通过分段将逻辑地址转换为线性地址，通过分页将线性地址转换为物理地址，逻辑地址由两部分构成，一部分是段选择器，一部分是偏移，段选择符存放在段寄存器中，如CS（存放代码段选择符）、SS（存放堆栈段选择符）、DS（存放数据段选择符）和ES、FS、GS（一般也用来存放数据段选择符）等
2. 偏移与对应段描述符中的基地址相加就是线性地址，操作系统创建全局描述符表和提供逻辑地址，之后的分段操作x86的CPU会自动完成，并找到对应的线性地址，从线性地址到物理地址的转换是CPU自动完成的，转化时使用的Page Directory和Page Table等需要操作系统提供

X86 和 MIPS 在内存管理上的区别

1. TLB不命中的处理：MIPS触发TLB缺失和充填，然后CPU重新访问TLB；x86硬件MMU索引获得页框号，直接输出物理地址，MMU充填TLB加快下次访问速度
2. 分页方式不同：一种MIPS系统内部只有一种分页方式；x86的CPU支持三种分页模式
3. 逻辑地址不同：MIPS地址空间32位；x86支持变长地址，逻辑地址为64位，同时提供转换为32位定址选项
4. 段页式的不同：MIPS同时包含了段和段页式两种地址使用方式，在x86架构的保护模式下的内存管理中，分段是强制的，并不能关闭，而分页是可选的

> **Thinking A.1** 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 $3 \times 9 + 12 = 39$ 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，若三级页表的基地址为 PT_{base}，请计算：

- 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）。

A.1

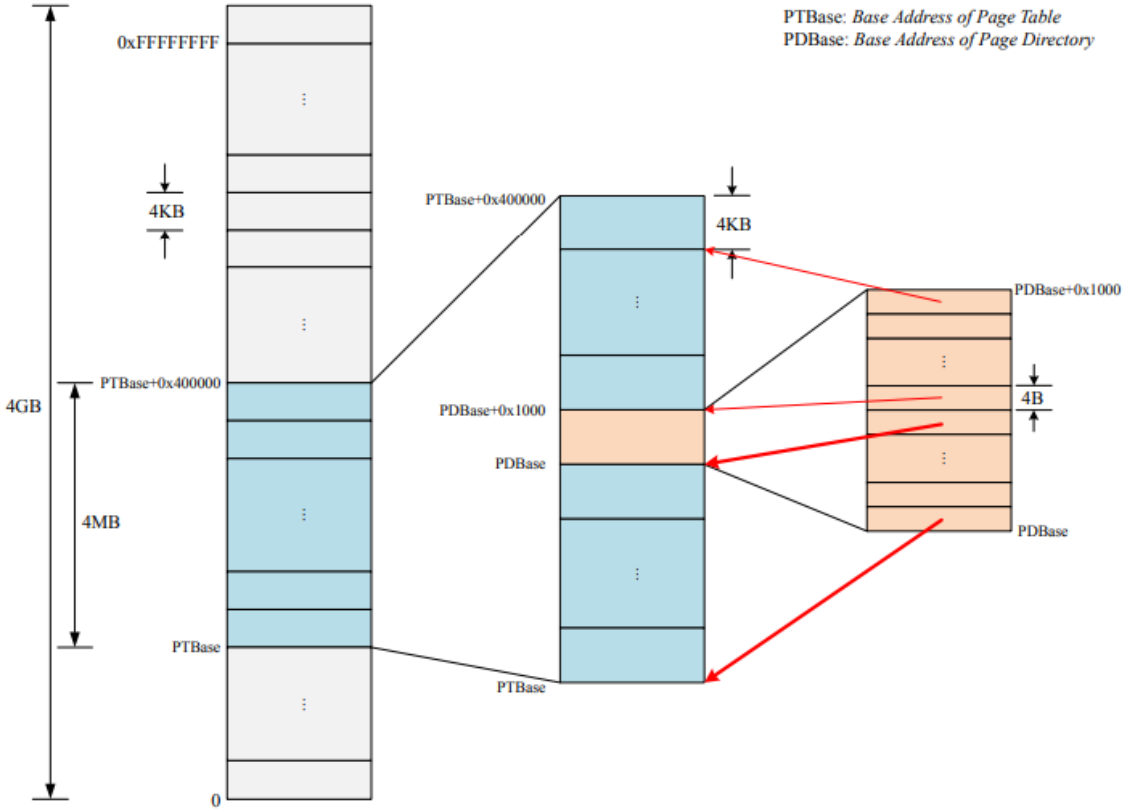
页目录自映射，在虚拟地址下，有以下映射关系

三级页表基地址 / 2^{30} = 二级页表基地址 / 2^{21} = 页目录基地址 / 2^{12} = 页目录项自映射的地址 / 2^3

解析：三级页表起始地址是 4G 中的第几个 1G，就等价于二级页表起始地址是 1G 中的第几个 2M，就等价于页目录表起始地址是 1G 中第几个 4K，最后也等价于发生自映射的那一条页目录项是 4K 中的第几个 8B，的因为从上到下的映射关系是一一对应的，后面的依次类推即可，对应虚拟地址的比例关系如下：

```
pte = (pte >> 30) << 30
△ppte = (pte >> 30) << 21
△pppte = (△ppte >> 21) << 12
△ppppte = (△pppte >> 12) << 3
```

真正的地址则是基地址加偏移量，图示如下：



exercise

系统宏函数解析

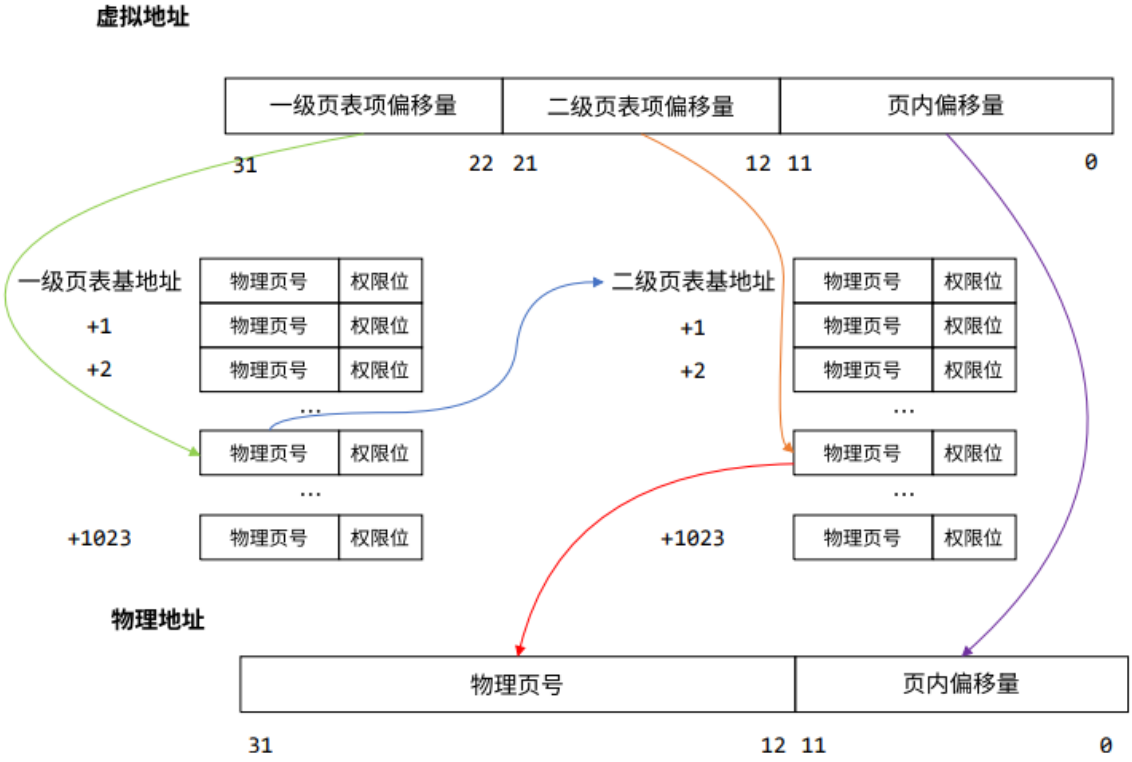
页目录项的结构：共有 32 位，高 20 位，表示这个页目录项指向的页表项的基地址所对应的物理地址，低 12 位，表示这个页目录项有关的控制位

页表项的结构：共有 32 位，高 20 位，表示这个页表项所对应的虚拟地址所转化成的物理地址，低 12 位，表示这个页表项有关的控制位

这里因为需要满足 4KB 对齐，因此从页目录中索引得到的高 20 位页表项基地址，以及从页表项索引得到的高 20 位内存的物理地址，都是不需要左移的，直接保持在高位即可，这是内存对齐特性所天然保证的，不需要我们再处理

对二级页表按需调入的理解：页目录地址是逻辑地址，但是这段逻辑地址中存储的地址索引确实页表基地址的物理地址

内核虚拟地址：内核虚拟地址是指内核代码访问内存时看到的地址，本质上就是虚拟地址，只是因为我们写的是内核代码，所以前面加上内核修饰，用于映射物理内存，供内核使用，也就是我们下面所有代码所操作的地址都应该是虚拟地址



在实验过程中，我们地址变换的过程如下：由页目录基地址 + PDX(va) 所得到的地址是 va 对应的页表项基地址的物理地址，对这个地址取一次 KADDR 则得到页表项基地址对应的逻辑地址，这个地址再加上 PTX(va) 就又得到这个 va 所对应的那个页表项的逻辑地址，最后取整个逻辑地址中所指向的地址即为 va 这个虚拟地址所对应的物理地址，其中最关键的一步映射是二级页表的**按需调入**，也就是通过页目录表索引到的页表基地址是一个**物理地址**，如果想用内核代码操作它，就必须先转化回逻辑地址才行，且这里我们转化的物理地址更严谨点说应该是**物理页面**地址，也就是低 12 位对齐的物理页桢

```
pgdir : 页目录基地址的虚拟地址;
pde : 表示一个页目录项(可能为页目录基地址)
pte : 表示一个页表项(可能为页表基地址)
pgdir + PDX(va) : va 所对应的页目录项的虚拟地址;
*(pgdir + PDX(va)) : va 所对应的页表项基地址的物理地址;
PTE_ADDR(*(pgdir + PDX(va))) : va 所对应的页表项基地址的物理地址只取高 20 位;
KADDR(PTE_ADDR(*(pgdir + PDX(va)))) : va 所对应的页表项基地址的逻辑地址;
KADDR(PTE_ADDR(*(pgdir + PDX(va)))) + PTX(va) : va 所对应的页表项的逻辑地址;
*(KADDR(PTE_ADDR(*(pgdir + PDX(va)))) + PTX(va)) : va 所对应的物理地址;
```

系统实现的函数或宏，以及一些变量命名习惯，在标程代码中我们需要用到它们：

```
系统定义的宏或变量名习惯:
// 和内存有关
u_long npage : 物理页数
u_long Pde : 页目录项类型(本质就是 unsigned long)
u_long Pte : 页表项类型(本质就是 unsigned long)
freemem : 初始变量，可用内存的起始地址，是虚拟地址(凡是内核代码操作的都是虚拟地址，比较越界时需要 PADDR 转化为物理地址);
memsize : 初始变量，总的物理地址大小，是物理地址
pages : 初始变量，存物理页的数组，每个 pages[i] 都是一个物理页;
page_free_list : 初始变量，Page_list 类型，作为空闲页链表的链表头，用来初始化空闲页链表;
PTE_V : 页表项的有效位，表示这个页面存在于物理内存中;
PTE_C_CACHEABLE : 页表项的可缓存标志位;
PDX(va) : 取虚拟地址 va 的高 10 位，作为页目录索引，也就是 va 在对应的页目录项中相对于页目录基地址的偏移量，这个偏移量本身的值大小在页目录的物理地址和虚拟地址中是等价的;
PTX(va) : 取虚拟地址 va 的中间 10 位，作为一级页表索引，也就是 va 在对应的一级页表中相对于页表基地址的偏移量，这个偏移量本身的值大小在所在页表的物理地址和虚拟地址中是等价的;
PPN(pa) : 由物理地址获取物理页号，也就是左移 12 位只保留高 20 位的结果
PDE* : 表示页目录基地址或者某个页目录项的地址(二级页表);
PTE* : 表示一级页表基地址或者某个页表项的地址;
Page* : 指向物理页结构体 Page 的指针;
Page : 内核中用来管理物理页的结构体，每一个物理页都有一个对应的结构体项，其中存储它在空闲页链表中指向下一个物理页的指针 pp_link，以及它的引用计数 pp_ref;
PTE_ADDR(*pgdir_entryp) : 传入一个地址，去掉低 12 位，只保留高 20 位，比如如果传入一个页目录项即这个页目录项所指向的一级页表项的基地址对应的物理地址左移 12 位的结果(12341234 -> 12341000);
KADDR(pa) : 把物理地址 pa 转化为内核虚拟地址，这是因为内核不能直接操作物理地址，需要通过映射访问;
PADDR(ka) : 把内核虚拟地址 ka 转化为物理地址;
page2pa(p) : 把 *p 指向的物理页框转化成对应的物理地址，物理页框是一个具体的物理页，物理地址用于页表 / 硬件中的操作;
pa2page(pa) : 把 pa 这个物理地址转化为它所对应的具体的物理页框 *p，进而用内核来管理这个具体的物理页;
ROUND(a, n) : 把某个数值向上对齐到某个倍数，如果页是 4KB 对齐，那么我们申请 4100 字节会变成 8192 字节;
page2kva(p) : 把物理页面转化为对应的内核虚拟地址(可以认为是 page2pa 是映射成物理地址，而 page2kva 直接映射到内核虚拟地址);
kva2page(k) : 把内核虚拟地址 kva 转换回它所对应的物理页 Page;
va2pa(va) : 把 va 这个虚拟地址转化成其对应的物理地址，在查页表的时候常用;
LIST_INIT(&list_head) : 链表操作宏函数，初始化一个链表，把链表头存入 list_head 中，整个链表为空;
LIST_EMPTY(&list_head) : 链表操作宏函数，检验以 list_head 为头的链表是否为空;
```

```
LIST_INSERT_HEAD(&list_head, new_node, field_name) : 链表操作宏函数，在链表头部插入节点，向 list_head 这个头指针所指向的链表中插入新节点 new_node， 其中 field 表示我们要操作的链表是哪一个，因为 list_head 这个结构体中可能包含多个链表字段，我们要选出我们要操作的那一个;
LIST_FIRST(&list_head) : 链表操作宏函数，返回链表第一个物理页面(不是物理地址，而是 Page* 页面)，如果链表为空，则返回 NULL;
LIST_REMOVE(node, field_name) : 链表操作宏函数，删除 node 节点，自动更新链表的前后指针，把对应节点摘除掉;
LIST_ENTRY(type) : 在结构体中声明链表指针子段，它会被展开成为 struct {

     struct type *le_next;  // 指向下一个节点
     struct type **le_prev; // 指向前一个节点的 next 指针的指针，这么做是为了方便删除节点
} 的形式;
asert() : 断言，判断一个表达式是否成立，如果不成立则输出错误;
pgdir_entryp : 变量名，指向页目录项(PDE)的指针，用来操作页表结构;
**ppte : 变量名，用来存某个虚拟地址 va 所对应的二级页表项的指针;
// 作业实现的函数
void mips_detect_memory(u_int _memsize) : 在内存中寻找 _memseize 这么大的地址空间，并且初始化物理总内存 memsize 的大小;
void *alloc(u_int n, u_int align, int clear) : 在内存中申请一块大小为 n 个字节，按照 align 字节对齐的地址空间，如果 clear 为 1，则要进行初始化;
void mips_vm_init() : 设置两级页表结构，实现虚拟地址到物理地址的映射;
void page_init(void) : 初始化物理页框，即初始化页面结构体，空闲也链表等结构;
int page_alloc(struct Page **new) : 从空闲页链表中找到一个物理页进行分配，得到的物理地址作为返回值，并且要用 new 指向这个物理页;
void page_free(struct Page *pp) : 释放 *pp 所指向的物理页，将其返回到空闲页链表中;
static int pgdir_walk(Pde *pgdir, u_long va, int create, Pte **ppte) : 查找 va 这个虚拟地址在二级页表中对应的页表项，其中 *pgdir 为页目录基址，va 为要映射的虚拟地址，create 是是否调入新的二级页表的标记，**ppte 用来指向 va 所对应的二级页表中的页表项;
int page_insert(Pde *pgdir, u_int asid, struct Page *pp, u_long va, u_int perm) : 把一个物理页面 *pp 插入到 *pgdir 作为基地址的页目录中，并且把虚拟地址 va 映射到这个物理页，其中 aisd 是地址空间标识符，perm 是页表项的权限标志;
struct Page *page_lookup(Pde *pgdir, u_long va, Pte **ppte) : 查找虚拟地址 va 所对应的物理页面，如果存在物理页面，就返回指向这个物理页面的指针，同时 *ppte 指向 va 所对应的二级页表项的指针;
void page_decref(struct Page *pp) : 减少 pp 对应的物理页面的引用计数一次，如果减为 0，则要把这个物理页放回空闲页链表中;
void page_remove(Pde *pgdir, u_int asid, u_long va) : 从页目录 pgdir 中删除 va 这个虚拟地址到物理页面的映射，其中 asid 为地址空间标识符;
// 和外存有关
da : 表示某个物理页在磁盘上的物理地址，也就是它被 swap out 到磁盘的时候，在磁盘中的地址，是页对齐的，低 12 位为 0，这一点和内存物理地址相同
PTE_SWP : 页表项中的交换位，置 1 表示这个页表项对应的物理页已经被 swap out 到外存
BY2PG :
page_free_swapable : 当前处于空闲状态，可以用来存放从外存中调入的页信息的物理页框列表，也就是当我们要从外存中 swap in 进来一个页面的时候，就要从 page_free_swapable 中找页面，用这个页面来存放换入页的内容；当某个页被换出到外存的时候，这个页的信息就不需要再维护了，那么当后面又有页被换入的时候，就可以用这个页来存储，因此此时又把它压入 page_free_list 队列中;
u_char *disk_alloc() : 类比 page_alloc，在磁盘空间中分配一个磁盘页大小的空间;
void disk_free(u_char *da) : 类比 page_free，清空地址为 da 的磁盘页的空间;
struct Page *swap_alloc(Pde *pgdir, u_int aisd) : 寻找一个可以用来存放外存中调入的页的信息的物理页框，先去 page_free_list 里面拿，如果没有空闲的，就强行把 PAGE_SWAP_BASE 这个页换出，然后用这个页的空间存我们换入的页的内容，本质上分配的是内存页;
static int is_swapped(Pde *pgdir, u_long va) : 判断某个虚拟地址对应的物理页是否被换出;
```

```
61 static void swap(Pde *pgdir, u_int asid, u_long va) : 把 va 这个虚拟地址对应的页表项中的磁盘地址中的信息，交换到内存中，同时更新页表对应磁盘地址的位置为物理地址;
```

注意：这里的我们所说(操作)的物理地址并不是完整的物理地址，而是物理地址中的高 20 位物理页桢(不左移)，因为我们是按页组织内存的，我们只需要得到每个页的起始物理地址，具体的页内偏移我们并不关心，这个物理页桢，就是页表项中的高 20 位(去掉低 12 位的标记位)，也就是每一个物理页转化回的物理地址，换言之，以下两种表述都是对物理页桢的描述：

```
PTE_ADDR(*pte);// 取页表项的高 20 位
page2pa(pp);// 把物理页转化为对应的物理页桢
```

有关知识点补充

（以下内容来自张杨学姐博客的内容）：

```
//已知Pde *pgdir
for (u_long i = 0; i < 1024; i++) { //遍历页目录的1024项
  Pde *pde = pgdir + i; //第i个页目录项对应的虚拟地址
  if (pde && ((*pde) & PTE_V)) { //第i个页表有效
    for (u_long j = 0; j < 1024; j++) { //遍历第i个页表的1024项
      Pte *pte = (Pte*)KADDR(PTE_ADDR(*pde)) + j; //第j个页表项对应的虚拟地址
      if (pte && ((*pte) & PTE_V)) { //第j个页有效
      /*do something...*/
      }
    }
  }
}

//已知Pde *pgdir
Pde *pde = pgdir + i;
   /*
      pde:                          第 i 个页目录项对应的虚拟地址
      (*pde):                       第 i 个页目录项内容
      PTE_ADDR(*pde):               第 i 个页目录项内容中的物理地址(去掉低 12 位的标记位) = 第i个页表的物理基地址
      KADDR(PTE_ADDR(*pde)):        第 i 个页表的虚拟基地址
  */
Pte *pte = (Pte*)KADDR(PTE_ADDR(*pde)) + j;
   /* pte:             第 j 个页表项对应的虚拟地址
      (*pte):          第 j 个页表项内容
      PTE_ADDR(*pte): 第 j 个页表项中的物理地址(物理页桢)
   */
u_long va =  (i << 22) | (j << 12) | offset ; // va 虚拟地址
u_long i  = PDX(va);
u_long j  = PTX(va);
```

课上：exam 内容

对于给定的页目录 pgdir，统计其包含的所有二级页表中满足以下条件的页表项：

1. 页表项有效;
2. 页表项映射的物理地址为给定的 Page *pp 对应的物理地址;
3. 页表项的权限包含给定的权限 perm_mask

code :

```c
u_int page_perm_stat(Pde *pgdir, struct Page *pp, u_int perm_mask) {
  Pte *pte;// 页表项地址
  Pde *pde;// 页目录项地址
  int cnt = 0;
  for (int i = 0; i < 1024; i++) {// 遍历所有页目录的项
    // 得到页目录中的第 i 个页目录项
    pde = pgdir + i;
    // 检验页目录项是否合法
    if ((*pte) & PTE_V) {
      for (int j = 0; j < 1024; j++) {// 遍历所有页表项
        // 得到第 i 个页目录项所对应的页表中的第 j 个页表项，强转 Pte* 只是为了形式上好看
        pte = (Pte *)KADDR(PTE_ADDR(*pde)) + j;
        // 检验页表项是否合法
        if ((*pte) & PTE_V) {
          // 检验对应权限 perm_mask 是否存在
          if (((*pte) | perm_mask) == mask) {
            // 检验物理地址是否和物理页 pp 等价
            if (PTE_ADDR(*pte) == page2pa(pp)) {
              cnt++;
            }
          }
        }
      }
    }
  }
  return cnt;
}
```

标程给出的 *code :*

```c
u_int page_perm_stat(Pde *pgdir, struct Page *pp, u_int perm_mask) {
  int count = 0;//统计满足条件的页表项的数量
  Pde *pde;
  Pte *pte;
  for (int i = 0; i < 1024; i++) {
    pde = pgdir + i;
    if(!(*pde & PTE_V)) { //当前页目录是否有效
      continue;
    }

    for (int j = 0; j< 1024;j++ ){
      pte = (Pte*)KADDR(PTE_ADDR(*pde)) + j;
      if (!(*pte & PTE_V)) { ////当前页表是否有效
        continue;
      }
      if (((perm_mask | (*pte))== (*pte))
                && (((u_long)(page2pa(pp)))>>12) == (((u_long)(PTE_ADDR(*pte)))>>12))
        count++;
            /*该层if判断条件等价于
            (perm_mask & (*pte))== perm_mask
            (page2pa(pp) == PTE_ADDR(*pte))
            */
    }
    }
  return count;
```

```
}
```

课上：extra 内容

实现从外存中调入一个页到内存，且当内存不够的时候，系统会把不常用的页 swap out 到外存，而当这个页被访问的时候，又会被 swap in 回到内存，现在需要我们模拟一个简化的页换入和换出的机制，在内存不足时只需要换出固定地址（0x39000000）的页面即可

无论是页的换入还是换出，我们都需要维护两个维度上的信息：

1. 内存本身：页换出的时候需要在磁盘中申请空间，拷贝内存到磁盘；页换入的时候需要在内存中申请空间，拷贝外存到内存
2. 页表信息：需要把虚拟地址对应的页表项的高 20 位在内存物理地址和磁盘地址之间转换，同时要更改标志位和有效位，做好 TLB 的无效化

code：

```
// 被动换出，当系统没有空闲页的时候触发
// 函数作用：找到一个可换出页，返回指向这个 page 的指针，如果不存在，就强行换出 SWAP_BASE_PAGE 这个物理地址对应的页

struct Page *swap_alloc(Pde *pgdir, u_int asid) {

  // Step 1: 如果不存在可交换页面，就需要进行页面换出操作
  if (LIST_EMPTY(&page_free_swapable_list)) {

    /* Your Code Here (1/3) */
    // 分配磁盘空间 disk_swap，并且把分配好的磁盘空间的地址记录在 da 中
    u_char  *disk_swap = disk_alloc();
    u_long da = (u_long)disk_swap;

    // 简化形式，只换出 0x39000000 的页，也就是  SWAP_BASE_PAGE 这个地址，我们把他对应的物理页存在 p 中
    struct Page *p = pa2page(SWAP_PAGE_BASE);

    // 遍历所有的页表项，找到页表中映射到 SWAP_BASE_PAGE 这个物理页面的那些页表项，这里的遍历套路和 exam 相类似，先遍历页目录，再遍历每个有效的页目录中那些有效的页表项
    for (u_long i = 0; i < 1024; i++) {
      Pde *pde = pgdir + i;
      if ((*pde) & PTE_V) {
        for (u_long j = 0; j < 1024; j++) {
          Pte *pte = (Pte*)KADDR(PTE_ADDR(*pde)) + j;
          if (((*pte) & PTE_V) && (PTE_ADDR(*pte) == SWAP_PAGE_BASE) ) {
            // 这里(da>>12)也可以写成(da/BY2PG)，其中 BY2PAGE 这个宏表示为每页的字节数，一般就是 4096，这条语句的含义是把 pte 这个页表项的内容中的高 20 位物理地址换成磁盘地址 da 的高 20 位，但是保持低 12 位标志位保持不变，(da>>12)<<12 的含义是取磁盘地址的高 20 位，把低 12 位清空，0xfff 是一个低 12 位全为 1 其余位全为 0 的数，*pte 和它相与即可只保留低 12 位，因此我们完成了替换
            (*pte) = ((da >> 12) << 12) | ((*pte) & 0xfff);
            // 这条语句的含义是我们需要把 pte 这个页表项对应内容的 PTE_SWP(换出标志位) 设成 1，PTE_V(在物理内存中有效标志位) 设成 0，我们只需要先或后与即可
            (*pte) = ((*pte) | PTE_SWP)  & (~PTE_V);
            // 这条语句的作用是使得 tlb 表项中 va 和 *pte 中原有的物理地址的映射关系失效，通过之前的分析，我们知道 tlb_invalidate 中的 tlb_out 函数其实只会调用 va 这个虚拟地址的高 20 位(不左移)，因此我们只需要把 va 的高 20 位，也就是 PDX 和 PTX 分别构造出来，他们就是页目录索引和页表项索引，分别左移 22 和 12 位即可得到
            tlb_invalidate(asid, (i << 22) | (j << 12) );
```

```
          }
        }
      }
    }

    // 把 va 这个虚拟地址对应的内容拷贝到磁盘
    memcpy((void *)da, (void *)page2kva(p), BY2PG);

    // 把交换出去的这个物理页重新变成可换出的页，也就是加入 page_free_swapable 链表中
    LIST_INSERT_HEAD(&page_free_swapable_list, p, pp_link);
  }

  // Step 2: 获取一个空闲页并清空
  // 从可换出页的链表中取第一个可换出页，取出以后把他从链表中删除，并且清空页面中的内容
  struct Page *pp = LIST_FIRST(&page_free_swapable_list);
  LIST_REMOVE(pp, pp_link);
  memset((void *)page2kva(pp), 0, BY2PG);

  // 返回我们取出的可换出页 pp
  return pp;
}

// 判断某个虚拟地址是否已经被换出
static int is_swapped(Pde *pgdir, u_long va) {

  // 根据页目录基地址 pgdir 和虚拟地址 va 查找对应的页表项
  /* Your Code Here (2/3) */
  Pde *pde = pgdir + PDX(va);
  if ((*pde) & PTE_V) {
    Pte *pte = (Pte*)KADDR(PTE_ADDR(*pde)) + PTX(va);

    // 如果当前页面设置了 PTE_SWP 并且没有设置 PTE_V，则说明它已经被换出了，其中 PTE_V 用来指示当前页是否在物理内存中，而 PTE_SWP 用来指示当前页是否在外存中，这两个标志位是互斥的，不能同时设为 1，但是因为如果 PTE_SWP 和 PTE_V 都为 0 的时候，未必表示页面已经换出，而是可能这个页既不在内存又不在外存，即尚未分配，因此需要把两重判断都加上
    if ((*pte & PTE_SWP) && !(*pte & PTE_V)) {
      return 1;
    }
  }
  return 0;
}

// 主动换入，将之前换出的页重新加载回内存
static void swap(Pde *pgdir, u_int asid, u_long va) {

  // 分配一个内存页，swap_alloc 就是我们之前填好的函数，它用于在 page_free_list 中找到一个可换出的页面 pp，并且将其清除后返回，让调用者可以添加新的内容进去，这个 pp 实质上是分配在内存里的空间，将用它来存放磁盘页的信息
  /* Your Code Here (3/3) */
  struct Page *pp = swap_alloc(pgdir, asid);

  // 找到页表项中的磁盘地址，实际上我们取出的是 va 这个虚拟地址对应的页表项的内容中的高 20 位，也就是 va 这个虚拟地址对应的物理页桢(物理地址高 20 位)，但是在 swap_alloc 函数中，我们对这 20 位做了修改，把物理页桢字段修改成了磁盘中的地址 da，并且打上了 PTE_SWP 标记表示这是一个已经换出到磁盘的页，同时取消了有效位 PTE_V，也就是以下两条操作：
  /*
  (*pte) = ((da / BY2PG) << 12) | ((*pte) & 0xfff);
  (*pte) = ((*pte) | PTE_SWP)  & (~PTE_V);
```

```
  */
  u_long da = PTE_ADDR(*((Pte*)KADDR(PTE_ADDR(*(pgdir + PDX(va))))) + PTX(va)));

  // 把磁盘内容复制到内存页
  memcpy((void *)page2kva(pp), (void *)da, BY2PG);

  // 遍历所有页表(1024 * 1024)，把所有映射到该磁盘页的页表项的高 20 位从磁盘地址再改回对应的物理页桢(物理地址的高 20 位)
  for (u_long i = 0; i < 1024; i++) {
    Pde *pde = pgdir + i;
    if (*pde & PTE_V) {
      for (u_long j = 0; j < 1024; j++) {
        Pte *pte = (Pte*)KADDR(PTE_ADDR(*pde)) + j;
        if (!(*pte & PTE_V) && (*pte & PTE_SWP) && (PTE_ADDR(*pte) == da)) {
          //以下三句话含义均可类比于 swap_alloc，需要我们修改 *pte 的内容，具体来讲就是把高 20 位设回 va 对应的物理页桢，低 12 位保持不变；权限位中 PTE_SWP 置 0，PTE_V 置 1，最后使得 va 在 TLB 中的原映射关系失效
          (*pte) = ((page2pa(pp) / BY2PG) << 12) | ((*pte) & 0xfff);
          (*pte) = ((*pte) & ~PTE_SWP) | PTE_V;
          tlb_invalidate(asid, (i << 22) | (j << 12) );
        }
      }
    }
  }

  // 最后释放磁盘页
  disk_free((u_char *)da);
  return;
}
```

标程代码涉及的两个结构体：

```
// Page 表示一个物理页
struct Page {
  struct Page *pp_link;
  u_short pp_ref;
}

// Page_list 表示空闲页链表中的一个节点
struct Page_list{
  struct {
    struct {
      struct Page *le_next;
      struct Page **le_prev;
    } pp_link;
    u_short pp_ref;
  } lh_first;
}

//我们可以认为 Page 是 Page_list 节点组成的一部分，我们把一个 Page 类型页面传入链表中，链表会先对其进行进一步包装获得一个真实的 Page_list 完整节点，再进行插入
```

课下：页表代码解析

以下是作业中需要我们填写的内存操作函数：

```
/* mips_detect_memory 中传入参数 ram_low_size 即可初始化下面列出的变量 */
static u_long memsize; /* 物理内存的总大小 */
u_long npage;          /* 可用的物理页面的数量 = 物理内存总大小 / 一个页面大小 */

Pde *cur_pgdir;       /*指向当前页目录（Page_directory）的指针，存储当前页表结构*/

struct Page *pages;   /*存储 Page 结构的数组，每个 page 都是一个物理页*/
static u_long freemem;/*记录当前空闲内存起始位置，初始时为 end，用来分配新的物理内存*/

struct Page_list page_free_list; /* 空闲物理也链表，用来管理未被使用的物理页 */

/* Overview:
  函数作用：根据传入的参数 _memsize，初始化物理总内存大小 memsize；根据一个物理页的大小 PGSHIFT 初始化物理页的数量 npage，打印内存分配信息
 */
void mips_detect_memory(u_int _memsize) {
/* Step 1: 赋值：_memsize -> memsize */
        memsize = _memsize;

/* Step 2: 计算 npage 的值，npage 是物理页数，也就是物理总内存可以被分成多少个 4kb 的物理页，用总内存除以 4kb 即可，也就是左移 12 位，这里我们默认的地址分配规则是 10 10 12，也就是一个页面大小是 12 位 */
/* Exercise 2.1: Your code here. */
////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////

        npage = memsize >> PGSHIFT;//通常 PGSHIFT 为 12，是一个宏定义常量

        printk("Memory size: %lu KB, number of pages: %lu\n", memsize / 1024, npage);//打印内存信息，前者以 kb 为单位，因此要除以 1024，后者是内存分成的物理页的数量
}


/* Lab 2 alloc：物理内存分配器 */
/* Overview:
  函数作用：根据传入的参数，分配 n 字节的物理内存，并且保证分配的地址满足 align 的对齐要求，用来实现页表管理时候的内存对齐要求，其中 align 是 2 的整数次幂，相当于 align 进制下的上取整
  clear 变量如果被标记为 1 传入，则需要在分配完物理内存以后，把分配的内存清零，注意这里是清 0，而不是释放，相当于 C 语言的 memset(arr, 0, n) 行为
  这个函数只在内核启动的时候，用来初始化虚拟内存系统
  这个函数自带一个分配后检验条件，如果可分配内存不足 n 字节，则系统调用 panic() 产生 error，进而终止运行
*/
void *alloc(u_int n, u_int align, int clear) {
        extern char end[];
//extern 外部变量关键字修饰，指向程序中最后一段代码或者全局变量的地址，end[] 是由链接器在编译时生成的，标记了内核或程序中已经使用过的内存的结束位置，这里虽然 end 是一个字符数组，但实际上存的就是一个地址，用足够大的整型是可以进行强转的
```

```
        u_long alloced_mem;    //用来保存分配的内存的起始地址，到时候返回的就是alloced_mem

/*
  freemem 表示可用的下一块内存的起始地址 / 空闲内存起始位置，如果第一次调用 alloc，它会被初始化为 end 所指向的位置，也就是链接器把内核代码和全局变量分配完以后，内存区域剩下部分的第一个地址
*/
        if (freemem == 0) {
                freemem = (u_long)end; // 需要类型强转
        }

/* Step 1: 确保 freemem 对齐到 align 字节边界上，ROUND 是用来实现字节对齐的宏函数，传入内存地址指针和需要对其的字节数，返回对齐好的内存指针，字节对齐是为了满足内存分配时地址满足硬件要求 */
        freemem = ROUND(freemem, align);

/* Step 2: 用 alloced_mem 保存当前 freemem 的值，用做函数的返回值 */
        alloced_mem = freemem;

/* Step 3: 更新 freemem 为指向下一段可用内存的地址，因为我们从当前 freemem 的位置开始，分配了 n 字节的空间，所以 freemem 应该指向 n 个字节以后的空闲位置 */
        freemem = freemem + n;

// 检查分配的内存地址是否超出了系统的物理内存范围。如果超出范围，就触发 panic_on 系统崩溃，PADDR 是一个宏函数，用来把 freemem 转化成对应的物理地址(这里 freemem 是虚拟地址，而我们分配的 memsize 是物理内存的范围，要比较二者需要先把 freemem 通过 PADDR 函数转化为物理内存)
        panic_on(PADDR(freemem) >= memsize);

/* Step 4: 如果 clear 参数被设成 1，则需要初始化我们申请好的内存空间为 0，此时这段空间的起始地址为 alloced_mem，大小为 n 字节，这里可以调用 memset 函数来实现 */
        if (clear) {
                memset((void *)alloced_mem, 0, n);
        }

/* Step 5: 返回我们申请好的内存的起始地址 */
        return (void *)alloced_mem;
}

/* Overview:
  设置虚存管理中的两级页表结构
  函数作用：分配和初始化物理内存，设置虚拟地址到物理地址的映射
 */
void mips_vm_init() {
/*
  为 pages 分配适当大小的物理内存，其中 pages 用于物理内存的管理
  把虚拟地址 UPAGES 映射到分配好的物理地址 pages
  内存分配的时候需要考虑内存对齐问题，因为分配的时候内存大小会向上取整
*/
// npage 为物理页数，PAGE_SIZE 为一个物理页的大小，调用 alloc 函数的含义为：申请大小为 npage * sizeof(page) 这么多字节的内存，需要满足对其 PAGE_SIZE，同时需要初始化，并且把申请好的内存转化为 Page* 类型
        pages = (struct Page *)alloc(npage * sizeof(struct Page), PAGE_SIZE, 1);
        //打印信息：申请了 freemem 这么多的内存作为物理页
        printk("to memory %x for struct Pages.\n", freemem);
        //打印信息：物理内存分配成功提示
```

```
        printk("pmap.c:\t mips vm init success\n");
}

/* Overview:
  初始化物理内存页管理系统
  函数功能：没有返回值和参数，负责初始化内存物理页结构和空闲页链表
 */
void page_init(void) {
/* Step 1: 初始化空闲内存页链表 page_free_list */
/* Exercise 2.3: Your code here. (1/4) */

//LIST_INIT 是一个宏函数，定义在 include/queue.h 中，可以初始化一个空的链表，他可以把传入的参数 page_free_list 链表初始化为一个空链表，它会修改 page_free_list，函数作用后 page_list_free 就作为空链表的链表头
        LIST_INIT(&page_free_list);

/* Step 2: 将当前可用内存地址 freemem 对齐到 PAGE_SIZE 的倍数 */
/* Exercise 2.3: Your code here. (2/4) */

        freemem = ROUND(freemem, PAGE_SIZE);

/* Step 3: 将 freemem 以下的所有内存页标记为已使用 (set `pp_ref` to 1) */
/* Exercise 2.3: Your code here. (3/4) */

// 要想知道 freemem 这个虚拟地址对应第几个物理页，先用 PADDR 把虚拟地址转化为物理地址，再用物理地址除以一个物理页的大小，即可得到已经使用的物理页的数量 size
        int size = PADDR(freemem) / PAGE_SIZE;
        int i;
       //遍历这些已经使用过的物理页，把他们的 pp_ref 子段设成 1，这个标记表示当前物理页已经被使用了
        for (i = 0; i < size; ++i)
        {
                pages[i].pp_ref = 1;
        }

/* Step 4: 标记剩下的物理页还没有被使用，范围就是从已经使用的物理页的下一页，到总的物理页数的这些物理页，标记分为两部分，一个是设置 pp_ref 为 0，一个是把这些页面插入到空闲页链表中 */
/* Exercise 2.3: Your code here. (4/4) */

        for (int i = size; i < npage; ++i)
        {
                pages[i].pp_ref = 0;
//链表的头插法，其中 LIST_INSERT_HEAD 为一个宏函数，作用是把 page[i] 这个物理页，插入到 page_free_list 这个链表的头部，而 pp_link 相当于是链表中的 *next 指针，它也是 Page_list 结构体中的一个字段，用来在链表中链接下一个物理页
                LIST_INSERT_HEAD(&page_free_list, pages + i, pp_link);//这里写成 pages[i] 也可以
        }
}
//////////////////////////////////////////////////////////////////////////////
//////////////////////////////////////////////////////////////////////////////
//////////////////////////////////////////////////////////////////////


/* Overview:
```

```
  函数作用：从空闲内存中分配一个物理页面，并且将该页面填充为 0，函数接受一个指向 Page* 的二级指针（这里传二级指针是为了修改一级指针指向的地址，相当于是指针本体的初始化）new 参数，new 用于接受分配好的页面地址
  如果分配失败，就返回 -E_NO_MEM，如果分配成功，就把分配好的 Page 的地址设成 *pp，并且返回 0
  注意：这个函数并不会自动增加 Page 中的 pp_ref 子段，如果调用者需要增加引用计数的值，必须自己显式地实现
 */
int page_alloc(struct Page **new) {
/* Step 1: 从可用内存 freemem 中获取一个页，如果失败，就直接返回 -E)MO_MEM，应该是一个错误类型 */
        struct Page *pp;// pp 用来暂时存储从空闲页面列表中取出来的页面
/* Exercise 2.4: Your code here. (1/2) */

        pp = LIST_FIRST(&page_free_list);//取出空闲页面列表的第一个页面
        if (pp == NULL)
        {
                return -E_NO_MEM;
        }

        LIST_REMOVE(pp, pp_link);
//pp就是当前链表的第一个元素，现在它已经被获取，需要从空闲页面链表中删除它，也就是调用 LIST_REMOVE 这个宏函数用来删除链表的头节点

        /* Step 2: 把获取到的物理页面内容初始化为 0，可以使用内存函数中的 memset */
        /* Exercise 2.4: Your code here. (2/2) */

        //其中 page2kva 是一个宏函数，用来把具体的物理页面转化为虚拟内存，然后再用 0 来进行初始化，因为这里用到内存操作函数，因此只能操作虚拟内存，所以用的是 page2kva 而不是 page2pa
        memset((void *)page2kva(pp), 0, PAGE_SIZE);

        *new = pp;//一级指针的赋值，用二级指针来初始化，这里写的很正确
        return 0;
}

////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////

/* Overview:
  函数功能：释放内存，释放 pp 所指向的页面，并将它标记为可用，通过设置 pp_ref 为 0 即可
 * Pre-Condition:
 *   'pp->pp_ref' is '0'.
 */
void page_free(struct Page *pp) {
        //使用 aeesrt 宏来检查 pp 的引用计数是否为 0，assert 是一种调试工具，它在调试模式下会检查给定的条件是否为真，如果条件为假，程序会中断并报告错误，如果页面 pp 的引用计数（pp_ref）不为 0，程序会在调试时终止并输出错误，这是为了确保页面在被释放之前没有其他地方引用它。引用计数为 0 意味着该页面不再被使用，因此可以安全地释放
        assert(pp->pp_ref == 0);
        /* 因为 pp 所指向的页面被释放，也就是变回可用页面，则把他插入到空闲页面链表的头部，调用头插函数即可 */
        /* Exercise 2.5: Your code here. */
        LIST_INSERT_HEAD(&page_free_list, pp, pp_link);
}
```

```
172 ////////////////////////////////////////////////////////////////////////////
    ////////////////////////////////////////////////////////////////////////////
    ////////////////////////////////////////////////////////////////
173
174 /* Overview:
175   函数作用：给定 pgdir 指向页目录基地址，pgdir_walk 返回虚拟地址 va 所对应的页表项的指
    针，其中 pgdir 是一个两级页表结构，ppte 用来指向我们所返回的对应的页表项的指针，这里也仍
    然是一个用二级指针初始化一级指针的样例
176   如果内存不足，则返回 -E_NO_MEM 错误参数，否则返回 0 表示申请成功，并且用 *ppte 表示
    取出的页表项
177  */
178 static int pgdir_walk(Pde *pgdir, u_long va, int create, Pte **ppte) {
179         Pde *pgdir_entryp;// Pde 是页目录类型，定义了一个页目录项指针
    *pgdir_entryp
180         struct Page *pp;// Page 是物理页面类型，定义了一个物理页指针 *pp
181
182 /* Step 1: 获取对应的页目录项，根据虚拟地址 va，计算对应那个的页目录项的地址，其中 PDX
    宏的作用是提取出虚拟地址中页目录索引的部分，pgdir 是页目录起始位置的指针，相当于页目录基
    地址，加上页目录索引，即可得到指向 va 对应页目录项的指针 pgdir_entryp */
183 /* Exercise 2.6: Your code here. (1/3) */
184
185         pgdir_entryp = pgdir + PDX(va);
186
187 /* Step 2:
188 1 检查对应的页表是否存在
189 2 首先检查页目录项是否有效，其中 PTR_V 是页表项中的有效位，如果页目录项无效就进入第一层分
    支
190 3 接着判断 create 参数，如果参数为真，就尝试分配一个新的物理页面，如果因为内存不足分配失
    败，由 try 函数来处理系统错误
191 4 接着把分配好的物理地址 *pp 通过 page2pa 这个宏获取以后，保存早页目录项中
192 5 最后设置页目录项的权限位，把缓存属性 PTE_C_CACHEABLE 和有效位 PTE_V 设置成为 1，表
    示该页表有效且缓存可用，同时标记这个物理页已经被使用了
193 6 如果不需要分配新的物理页，则直接把 ppte 设为 NULL，表示没有找到合法的页表项
194 */
195 /* Exercise 2.6: Your code here. (2/3) */
196
197         if (!((*pgdir_entryp) & PTE_V))// 1 & 2
198         {
199                 if (create)// 3
200                 {
201                         try(page_alloc(&pp));// 3  这里用 try 包装，自带错误检
    查，如果分配页面失败，则自动报错
202                         *pgdir_entryp = page2pa(pp);// 3
203                         *pgdir_entryp = (*pgdir_entryp) | PTE_C_CACHEABLE |
    PTE_V;// 4
204                         pp->pp_ref++;
205                 }
206                 else
207                 {
208                         *ppte = 0;// 6
209                         return 0;
210                 }
211         }
212
213 /* Step 3: 获取页表项并且进行返回，这里返回的是虚拟地址 */
214 /* Exercise 2.6: Your code here. (3/3) */
215 /*
216   解析：
```

```
  1.pgdir_entryp 本身是一个页目录项的虚拟地址，但 *pgdir_entryp 中存储的则是这个页目录项所对应的页表项基地址的物理地址，这是因为由页目录项索引得到的页表基地址是物理地址(页目录项按位拆成高 20 位为它索引的页表项的基地址，低 12 位为标记为，但是因为要满足 4KB 的地址对齐，因此它索引得到的 20 位页表项基地址要 << 12，也就是仍然保持在高位而不需要左移)
  2.需要区分 pgdir_entryp 和 *pgdir_entryp 的值，二者含义不同
  3.PTE_ADDR(*pgdir_entryp) 表示提取该物理地址的高 20 位，也就是页表基地址的物理地址，因为每个页表页的大小都是 4KB，所以低 12 位都是全 0 对齐的状态，我们可以只提取出高 12 位来处理
  4.KADDR(PTE_ADDR(*pgdir_entryp)) 表示把页表基地址的物理地址转化回逻辑地址
  5.KADDR(*pgdir_entryp) + PTX(va) 表示当前页表项基地址的逻辑地址加上页表索引(PTX)，那么就得到了当前页表项相对于页表基地址的虚拟地址
  6.所以最后返回的是 va 这个虚拟地址所对应的页表项在虚存中的逻辑地址
  7.注意：为什么不能先加偏移量，再整体转化成虚拟地址，我认为这是由当前内存组织关系下的 KADDR 这个函数决定的，它只能把每个页表页的基地址物理地址正确的映射回虚拟地址，而页表页中的其他物理地址在加上偏移量以后，就无法正确地映射回虚拟地址了，因此只能先转成虚拟地址再加偏移(?)
*/
        *ppte = (Pte *)KADDR(PTE_ADDR(*pgdir_entryp)) + PTX(va);
        return 0;
}

//////////////////////////////////////////////////////////////////////////////
//////////////////////////////////////////////////////////////////////////////
//////////////////////////////////////////////////////////////////////////////

/* Overview:
  函数作用：把物理页面 *pp 映射到虚拟地址 va，并设置页表项有关权限（地址的低 12 位）应该设置成为 perm | PTE_C_CACHEABLE | PTE_V
  如果分配成功就返回 0，否则返回错误参数 -E_NO_MEM
  如果 va 已经映射了一个页面，则调用 page_remove() 释放这个映射，如果插入成功，就把 pp_ref 引用计数 +1
  入参含义：指向页目录的指针 *pgdir，地址空间标识符 asid（用来区分不同的地址空间），指向要映射的物理页面的指针 *pp，待映射到的虚拟地址 va，页面权限设置参数 perm
 */
int page_insert(Pde *pgdir, u_int asid, struct Page *pp, u_long va, u_int perm) {
        Pte *pte;

        /* Step 1: 获取 va 这个虚拟地址对应的页表项，上一个函数我们已经实现了，返回值存在 Pte 页目录类型的 pte 指针中，且 create 参数为 0，说明如果找不到 va 这个虚拟地址，不需要我们手动创建一个新页面 */
        pgdir_walk(pgdir, va, 0, &pte);
        //判断是否已经有页表项的映射：即 pte 存在，且页表项对应的 PTE_V 参数有效，说明已经存在映射
        if (pte && (*pte & PTE_V)) {
          //检查当前映射的物理页面是否和要插入的物理页面 pp 相同，如果不同则需要重新映射，在应设置前应该先释放掉当前的映射，使用 page_remove 进行释放；否则，如果页面已经正确映射，那么通过 tlb_invalidate 函数刷新 TLB，然后更新页表项，通过 page2pa 获取物理地址，并且把对应权限设置好
                if (pa2page(*pte) != pp) {
                        page_remove(pgdir, asid, va);
                } else {
                        tlb_invalidate(asid, va);// TLB 刷新
                        *pte = page2pa(pp) | perm | PTE_C_CACHEABLE | PTE_V;
                        return 0;
                }
        }
```

```
253
254         /* Step 2: 使用 tlb_invalidate 这个函数刷新 TLB 快表 */
255         /* Exercise 2.7: Your code here. (1/3) */
256
257         tlb_invalidate(asid, va);
258
259
260         /* Step 3: 重新获取或创建页表项，这里仍然是调用 pgdir_walk 函数，且此时如果页表项不存在我们需要创建一个新的页表项进行返回 */
261         /* 如果创建失败，则需要报错 */
262         /* Exercise 2.7: Your code here. (2/3) */
263
264         try(pgdir_walk(pgdir, va, 1, &pte));
265
266
267         /* Step 4: 把新创建好的页面插入到页表项，首先通过 page2pa 把 *pp 这个物理页面结构体指针转化为一个真正的物理地址，接着通过 perm PTE_C_CACHEABLE PTE_V 这三个参数设置相关的标记为，最后用 *pte 指向我们创建好的页面，插入页表项中 */
268         /* Exercise 2.7: Your code here. (3/3) */
269
270         *pte = page2pa(pp) | perm | PTE_C_CACHEABLE | PTE_V;//传入标记，可读写标记，有效位标记
271         pp->pp_ref++;//引用计数 +1，表示当前物理页面已经有了一个虚拟地址的映射
272
273         return 0;
274 }
275
276 ////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
277
278 /* 查找虚拟地址 va 所映射到的物理页面 */
279 /*Overview:
280   函数功能：在 *pgdir 所指向的页目录中，查找 va 这个虚拟地址所指向的物理页面，把查找到的结果存在 *ppte 中，也是一个用二级指针初始化一级指针的实例
281   在存入 *ppte 的同时，也要返回我们找到的这个物理页面所转化成的物理地址；如果找不到就返回 NULL（物理页面结构体指针和物理地址是两回事，后者存在 不 NULL 的 *ppte 中，前者是函数返回值，根据教程组的习惯，标记 page 的是结构体指针，标记 pa 的是物理地址）
282 */
283 struct Page *page_lookup(Pde *pgdir, u_long va, Pte **ppte) {
284         struct Page *pp;
285         Pte *pte;
286
287         /* Step 1: 调用 pgdir_walk 函数，查找虚拟地址 va 所对应的页表项，把他存入 pte 当中 */
288         pgdir_walk(pgdir, va, 0, &pte);
289
290         /* 如果没有找到对应的页表项，或者找到了但是页表项的有效位为 0，则返回 NULL，说明 va 不存在某个物理地址与之映射 */
291         if (pte == NULL || (*pte & PTE_V) == 0) {
292                 return NULL;
293         }
294
295         /* Step 2: 通过 pa2page 把页表项的物理地址转化为物理页面结构体指针，这个函数应该是 page2pa 的逆函数，这两个函数都是系统为我们实现好的函数，我们直接调用即可 */
296         /* 提示：page2pa，pa2page 中的 2 应该是 to 的简写 */
297         pp = pa2page(*pte);
```

```
        //这里我们做了一个检验，调用者传入的页表项指针 ppte 必须保证不为 NULL，此时我们才能把映射到的物理地址赋值给它，但是不管它 NULL 与不 NULL，我们都会把这个地址返回
        if (ppte != NULL) {
                *ppte = pte;
        }

        return pp;
}

/* Overview:
  函数作用：通过减少一个物理页面 pp 的引用计数 pp_ref，当引用计数减少到 0 的时候，表示不再有任何虚拟地址来引用这个物理页面，此时函数会调用 page_free 来释放这个页面
 */
void page_decref(struct Page *pp) {
        assert(pp->pp_ref > 0);

        /* 如果引用计数减少到 0，注意调用这个函数本身还要减一次，那么就释放这个物理页面 */
        if (--pp->pp_ref == 0) {
                page_free(pp);
        }
}

/* 取消映射关系 */
/* Overview:
  函数作用：取消虚拟地址 va 和他对应的物理页面的映射关系
  入参含义：pgdir 指向页目录（Page Directory）的指针，asid 是地址空间标识符，用来区分不同进程的虚拟地址空间，TLB 刷新函数需要这个参数，我们实现的时候不需要它，va 是待处理的虚拟地址
*/
void page_remove(Pde *pgdir, u_int asid, u_long va) {
        Pte *pte;

        /* Step 1: 首先找到虚拟地址 va 对应的物理页面，存在 pp 中，同时 pte 中存这个物理页面转化成的物理地址，如果 va 本身都没映射到哪个物理地址，就直接返回就好 */
        struct Page *pp = page_lookup(pgdir, va, &pte);
        if (pp == NULL) {
                return;
        }

        /* Step 2: 减少一次物理页面 pp 的引用计数 */
        page_decref(pp);

        /* Step 3: 更新 TLB，就是这个函数用到 asid 参数，所以我们在外面还要传进来 */
        *pte = 0;
        tlb_invalidate(asid, va);
        return;
}

//下面两个函数就没有需要我们填写的内容了，他们是用来检验函数正确性的内容，与核心代码的实现关系不大，这里不再解析
```

课下：快表代码解析

有关 TLB：

我们维护的 TLB 表结构分别存储在 entryHi，entryLo0，entryLo1这三个寄存器中，这三个寄存器都是 CP0 中的寄存器，并不是 TLB 表本身，他们只是对应到 TLB 中的 key 值和两组 data 值，TLB 本身只有 26 位，其中高 20 位为物理页(PFN)，低 6 位为权限位

我们在维护 TLB 的时候，按照 entryLo0 中存偶数页，entryLo1 中存奇数页的形式进行组织，而 TLB 的内容填写来自于页表查找，必须保证我们读出的页表项是偶数页在前，奇数页在后，首先易知，页表项的地址组织是**4字节**对齐的(因为一个页表项大小为 4 字节)，也就是说页表项的虚拟地址的低 2 位一定为 0，但是我们填写 TLB 的时候，要求一次填好两个 TLB 的表项，也就是先偶后奇的 8 字节对齐，又因为所有页表项是从 0 号开始组织的，也就是 01 23 45 67 89……这样每两个表项为一组，所以需要保证 8 字节对齐，且对齐地址的起始位置是偶数编号的页面，也就是需要保证低 3 位都是 0

查找到**页表**(32位)如何转化为**TLB**(26位)，只需要舍弃掉低 6 位即可，也就是把对应页表对应的地址左移 6 位即可

entryHi 的结构：高位为虚拟页号(VPN)，中间位置空，低位为地址标识符(ASID)

entryLo：分为奇数页和偶数页，分别存在 entryLo0 和 entryLo1 中，MIPS 采用的是**双页映射**的 TLB 结构方式，一个 TLB 条目对应两个连续的虚拟页(奇页 / 偶页)，分别映射到的虚拟页号是 2 × VPN 和 2 × VPN + 1

entryLo 的结构：高的 1 位保留，中间位为 PFN (物理页的基地址 24 位)，C 位为缓存策略(3 位)，D 位为脏位(1 位)，V 位为有效位标记(1 位)，G 位为全局条目标记(1 位)

TLB 事实上构建了一个映射 < VPN, ASID > ->(TLB)-> < PFN, N, D, V, G >

tlb 有关操作：kern / tlbex.c

我们修改操作的都是 CP0 中的三个寄存器，要把他们的变化实时映射到 TLB 表项中，就需要下面的 TLB 操作函数来完成：

```
有关指令 ：
tlbr ：(读取)以 index 寄存器中的值为索引，读出这个索引对应的 TLB 表项，并把表项内容存入 entryHi 和 entryLo
tlbwi ：(index)以 index 寄存器中的值为索引，把 entryHi 和 entryLo 中的值写入到 index 索引所对应的 TLB 表项中
tlbwr ：(random)把 entryHi 和 entryLo 寄存器中的数据随机写入到一个 TLB 表项中(使用 Random 寄存器循环技术得到编号)
tlbp ：(探测)根据 entryHi 寄存器中的 key 值(包括 VPN 和 ASID)找到 TLB 中与之对应的表项，并且把表项索引存入 Index 寄存器，如果未能找到，则 Index 高位置 1
有关寄存器 ：
CP0_INDEX ：存目标 TLB 表项的 index 索引
CP0_ENTRYHI ：存目标 TLB 表项的 entryhi 中的 key 值
CP0_ENTRYLO0 ：存目标 TLB 表项的偶页中的 val 值
CP0_ENTRYLO1 ：存目标 TLB 表项的奇页中的 val 值
```

这两段代码通过 tlb_invalidate 这个函数联系起来，只要我们的页表发生了改变，我们就需要调用 tlb_invalidate 函数对 TLB 进行刷新：

```
更新了某个虚拟页的物理映射；
撤销了某个虚拟页的映射；
更改了某个页的权限；
```

标程代码实现：

这里更加说明了，其实我们调用 page_insert 时候传入的 va 地址是否要把低 12 位设成 0 是都可以的，事实上我们需要这么做，但是每个函数内部都会有低位清零的操作，所以我们只是不需要额外在函数入口处手动清一次低位

```
/* tlb_invalidate 核心代码 */
/* Overview:
  函数作用：使得指定 asid 和 虚拟地址 va 所对应的 TLB 表项失效，具体实现为构造一个新的 Entry HI 并调用 tlb_out 来刷新 TLB
 */
void tlb_invalidate(u_int asid, u_long va) {
// GENMASK(PGSHIFT, 0) 表示生成一个从 0 位到 PGSHIFT 位都为 1 的掩码，取反以后变成高位全是 1，但是低位全是 0 的一个掩码，把 va 和这个取反后的掩码按位与以后可以把 va 的低 12 位置成 0，也就是实现页大小对齐；NASID 表示为系统支持的 asid 最大值，通常是一个全 1 的掩码，则 asid 和 NASID-1 取与运算以确保 asid 在有效范围内；最后把这两部分按位或即可得到一个完整的 TLB 索引地址，tlb_out 中传入一个合法的 tlb 索引值，这个函数就是使得这个索引值变得失效
        tlb_out((va & ~GENMASK(PGSHIFT, 0)) | (asid & (NASID - 1)));
}
/* 补充说明：
  当操作系统修改了页表（如取消映射、更改权限等）时，需要确保 TLB 中的旧缓存被清除，否则可能导致内存访问不一致
  在进程切换时，可能需要刷新旧进程的 TLB 条目
*/

/* overview
  函数作用 ：在发生缺页异常的时候，对虚拟地址 va 所对应的物理页进行被动分配，也就是在页表中建立 va 和物理页之间的映射
  入参含义：发生缺页的虚拟地址 va，当前进程的页目录基地址，地址空间标识符
*/
static void passive_alloc(u_int va, Pde *pgdir, u_int asid) {
        struct Page *p = NULL;

// 判断地址合法性
        if (va < UTEMP) {
                panic("address too low");
        }
        if (va >= USTACKTOP && va < USTACKTOP + PAGE_SIZE) {
                panic("invalid memory");
        }
        if (va >= UENVS && va < UPAGES) {
                panic("envs zone");
        }
        if (va >= UPAGES && va < UVPT) {
                panic("pages zone");
        }
        if (va >= ULIM) {
                panic("kernel address");
        }

        panic_on(page_alloc(&p));
// 权限位的设置是看是否可写，如果在 [UVPT,ULIM) 范围内就不给写权限，反之就赋予写权限 PTE_D，同时要建立物理地址和虚拟地址之间的映射关系，这里取出的是 va 的页对齐地址，也就是清除低 12 位的结果，其实这里清不清都是无所谓的
        panic_on(page_insert(pgdir, asid, p, PTE_ADDR(va), (va >= UVPT && va < ULIM) ? 0 : PTE_D));
}

```

```c
/* Overview:
  函数功能：重新填写 TLB 表
  入参含义：pentrylo 用于输出 entryLo0 和 entryLo1 的值，va 是触发异常的虚拟地址，asid 是地址空间标识符，用来支持多个进程共存
 */
void _do_tlb_refill(u_long *pentrylo, u_int va, u_int asid) {
        tlb_invalidate(asid, va);//先让旧的表项失效，保证更新 TLB 的时候不会有冲突
        Pte *ppte;
/*
  循环调用 page_lookup 查找当前当前进程的页目录 cur_pgdir 中虚拟地址 va 对应的页表项 *ppte
  如果 page_lookup 返回 NULL（表示未找到页表项），则通过 passive_alloc 分配新页，直到 page_lookup 成功
  查不到就分配，直到能查到为止
*/
/* Exercise 2.9: Your code here. */

        while (page_lookup(cur_pgdir, va, &ppte) == NULL) {//这里如果能找到，pte 中存的就是 va 这个虚拟地址对应页表项的虚拟地址
                passive_alloc(va, cur_pgdir, asid);
        }

// 构造 TLB 表项：TLB 要求的数据是虚拟地址为 va & ~0x1 的偶数页和它的下一个页，也就是偶数页+奇数页的形式，也就是 ppte[0] 和 ppte[1] 二者必须分别是偶数页和奇数页，但是我们取出的 ppte 本身可能指向一个奇数页(所谓偶数页，就是对应虚拟地址的最低位为 1，所谓偶数页，就是对应虚拟地址的最低位为 0)，那么我们取 ppte[0] 和 ppte[1] 就得到奇数页+偶数页的形式，这是不符合顺序要求的，所以必须先退回到以偶数页作为起始地址，而一个页表项占 4 字节，两个页表项就是 8 字节，所以我们需要把 ppte 回退到偶数对齐的起点，也就是做到 8 字节对齐

// 这里要区分页号和页表项虚拟地址的区别，页号是连续的，页表项虚拟地址是按 4 字节增加的，因为一个表项的大小为 4 字节，而 TLB 读取时要求我们每次读出两个表项，也就是一次读出 8 个字节，又因为页号是从 0 开始组织的，第一次读到的表项一定是偶数页，所以我们必须保证地址一直是 8 字节对齐，则每次读出来的两个页表项都是偶数页+奇数页的形式
        ppte = (Pte *)((u_long)ppte & ~0x7);

// ppte[0] 是当前虚拟地址 va 对应的页表项，而 ppte[1] 是这个页表项紧邻的下一个页表项，二者分别是偶数页和奇数页，且为 32 位，但是 TLB 只需要 26 位，也就是我们需要舍弃对应页表项的低 6 位，所以进行左移即可 ，这里我们要区分页号和页虚拟地址，前者每增加一个，后者则增加四个字节
        pentrylo[0] = ppte[0] >> 6;
        pentrylo[1] = ppte[1] >> 6;
// 这里我们向 entryLo 寄存器中写入的是物理地址，也就是 ppte 和 ppte + 4(字节) 这两个虚拟地址对应的页表中索引得到的物理地址，又因为 *(a + i) 和 a[i] 二者是等价的，同时指针 +1 跳过的是一个结构体的步长，因此这样恰好可以取出对应的物理地址
}
```

思考题：tlb_out 函数解析

kern / tlb_asm.S：与 tlb **失效**和**重写**有关的函数

```c
// tlb_out(a0) ：用于从 TLB 中清除 a0 对应的虚拟地址的映射，先查找是否存在，如果存在就清除，否则直接返回
LEAF(tlb_out)// 宏定义，说明 tlb_out 为叶子函数，不会调用其他函数
.set noreorder// 禁止编译器重新排列指令，保证指令顺序不变

// 将原始的 entryHi 保存在 t0 寄存器中，把传入的虚拟地址从 a0 写入 entryHi 中用于 TLB 查询
```

```
        mfc0    t0, CP0_ENTRYHI
        mtc0    a0, CP0_ENTRYHI
/* Step 1: 使用 tlbp 来探测当前 CP0_ENTRYHI 指定的虚拟地址是否已经存在于 TLB ，查找结果存入 CP0_INDEX，如果查找命中则 CP0_INDEX 会设置为查找到的 TLB 的 index，如果未命中则 CP0_INDEX 的最高位会设成 1 */
/* Exercise 2.8: Your code here. (1/2) */

        nop
        tlbp
        nop

/* Step 2: 根据查找结果做条件跳转 */
        mfc0    t1, CP0_INDEX // 把查找结果读取到 t1 寄存器
.set reorder
        bltz    t1, NO_SUCH_ENTRY // 如果 t1 为负，说明最高位为 1，则说明查找 TLB 没有命中，跳转到 NO_SUCH_ENTRY
.set noreorder
// 如果没有跳转，则说明查找到对应的 TLB，则执行下面的指令
        mtc0    zero, CP0_ENTRYHI // 把 0 写入 entryHi，相当于清除 entryHi 中的虚拟地址
        mtc0    zero, CP0_ENTRYLO0 // 把 0 写入 entryLo0，相当于清除 entryLo0 中的偶页映射
        mtc0    zero, CP0_ENTRYLO1 // 把 0 写入 entryLo1，相当于清除 entryLo1 中的奇页映射
        nop
/* Step 3: 使用 tlbwi 函数把修改后的 entryhi，entrylo0，entrylo1 分别写入 CP0_INDEX 指定的 TLB 表项中  */
/* Exercise 2.8: Your code here. (2/2) */

        tlbwi

.set reorder

// 没有命中 TLB 时的分支，恢复旧的 entryHi 寄存器，并且直接返回即可
NO_SUCH_ENTRY:
        mtc0    t0, CP0_ENTRYHI
        j       ra
END(tlb_out)


// do_tlb_refill ：在 TLB 缺失的时候，需要经过页表查找，把新映射填入 TLB 条目中

// 获取 TLB 缺失时的地址和当前地址空间信息
NESTED(do_tlb_refill, 24, zero) // 函数开头，分配 24 字节栈空间，函数传入参数分别为引发缺失的 va 和 TLB 的 entryHi
        mfc0    a1, CP0_BADVADDR // CP0_BADVADDR 存放引发 TLB 缺失的虚拟地址
        mfc0    a2, CP0_ENTRYHI // 向 CP0 中写入当前 TLB 的 entryHi
        andi    a2, a2, 0xff // 提取 ASID 的低 8 位

// 调用实际页表查找函数 _do_tlb_refill
.globl do_tlb_refill_call; // 全局符号确保可以外部调用
do_tlb_refill_call:

// 这里首先分配了 20 字节的空间，最高的 4 位用来存 ra，中间的 8 位依次存 pte0 和 pte1，但是这里没有把二者存到 sp 栈空间中，只是在栈中给 a0 返回值预留了地址，分别把 [12, 15] 和 [16, 19] 这两段地址从下到上给了 pte0 和 pte1 这两个返回值，因此取的顺序是先取 pte0 在下，后取 pte1 在上
        addi    sp, sp, -24 /* 分配 24 字节的栈空间 */
```

```
        sw      ra, 20(sp) /* [sp + 20] - [sp + 23] 保存函数返回地址 */
        addi    a0, sp, 12 /* [sp + 12] - [sp + 19] 设置 a0 作为返回值空间 */
        jal     _do_tlb_refill /* 调用页表处理函数，返回 PTE0 / PTE1 */

// 读取页表查找结果，准备写入 TLB
        lw      a0, 12(sp) /* 读取 PTE0 偶页 */
        lw      a1, 16(sp) /* 读取 PTE1 奇页 */
        lw      ra, 20(sp) /* 恢复返回地址 */
        addi    sp, sp, 24 /* 恢复栈空间 */

// 把 ao 和 a1 信息写入 entryLo0 和 entryLo1，并且调用 tlbwr 随机写入 TLB

        mtc0    a0, CP0_ENTRYLO0 /* 写入偶页 */
        mtc0    a1, CP0_ENTRYLO1 /* 写入奇页 */
        nop
/* 使用 tlbwr 随机找一个 TLB 进行写入，并最终返回 */
/* Exercise 2.10: Your code here. */

        tlbwr
        jr      ra
END(do_tlb_refill)
```