Ausar写的指导书

> 如果发现有错漏，请及时联系本人，或课程组老师和其他助教

lab2

相关知识介绍

内存地址的划分与映射

在32位的MIPS CPU中，由于地址只能由32位的二进制数表示，同时内存的最小单位是1字节（Byte），因此其最大寻址的空间为$2^{32} * 1Byte = 4GB$。

然而，这4G的空间并不是能直接使用的。提供给CPU的地址并不一定是实际访问物理内存的地址，因此又叫做**虚拟地址**。虚拟地址按照相应的规则转换之后才能得到真正的地址，即**物理地址**。

这4G的虚拟地址空间，被划分为4个不同的空间。每个空间有自己的转换规则

- kuseg(`0x0000_0000` ~ `0x7fff_ffff`)(大小为2G)

  这一段是用户模式下可用地址，经过缓存。

  如果CPU访问了这一段地址，假设这个地址为X，CPU并不会直接访问到物理内存上地址为X的这一段数据，而是要经过**MMU**的转换。

  可以形象地认为MMU为一个函数，假设MMU(X)=Y,则实际上CPU访问的是物理地址为Y的内存。

- kseg0(`0x8000_0000` ~ `0x9fff_ffff`)(大小为512M)

  这一段是内核地址，经过缓存。

  这一段的地址，同样地需要经过转换才能得到真正的物理地址。

  但是不需要经过复杂的MMU转换，而是**直接减去一个偏移** `0x80000000`。也就是说，会被直接映射到`0x0000_0000`~`0x1fff_ffff`的物理地址上

  在实际操作中，直接**清空地址二进制的高一位**（也就是按位与上`0x7fff_ffff`）即可完成该操作。

  （同学可以自己用二进制的方式推导一下这么操作的原理）

- kseg1(`0xa000_0000`~`0xbfff_ffff`)（大小为512M）

  这一段地址是内核地址，不经过缓存

  与kseg0类似，这一段地址向物理地址转换的方法也同样是清空虚拟地址的**高三位**即可完成转换。映射到物理地址的低512M区域中。(即，按位与上`0x1fff_ffff`)

  这一段内存不经过缓存，所以适合用来映射到硬件的I/O寄存器中，以避免缓存带来的一致性问题。但是如果用于其他用途，则会因为没有缓存导致速度较慢

- kseg2(`0xc0000000`~`0xffffffff`)(大小为1G)

  这一段地址是内核地址，经过缓存。且需要通过MMU进行转换。

MMU与页式内存管理

前文中提到了一个模块叫做**MMU**，这个模块是用于将虚拟内存地址转换为物理内存地址用的。但是具体是如何转换的呢？为什么需要对内存进行转换呢？

为什么需要内存管理技术

假设小明的电脑比较破，只有4MB的内存。他要运行一个程序，这个程序只需要4KB的内存即可运行。但是这个程序写得比较丑，居然访问的内存是在 `0x00f0_0000` 这个位置，这个位置对应的是内存中的第15MB。

那么这个时候就比较尴尬了，明明这个程序只用到4KB的内存，小明的电脑理论上来说是可以运行的，但是他访问的地址又超过了最大物理地址的范围，所以并不能运行。

那么怎么解决这样的问题呢？

机智的你应该能想出来，如果我们把程序所需要的这4KB内存**映射**到某一处物理内存上，那么岂不妙哉？

也就是说，如果我们能设计一种合理的内存映射机制，那么我们就可以让程序认为自己有**4G的内存空间随意访问**

（前提是，总的内存用量不能超过物理内存）

并且，通过内存映射，可以实现内存共享，也就是说两个不同的内存地址，实际上对应的都是同一个物理内存。就方便各个程序之间进行数据的交互，或者是运行多个相同程序的时候，只需要加载一遍代码到内存中，不需要白白浪费内存空间。

而在不同的程序中，很可能恰好访问到同一个内存地址。而此时我们希望这两个程序之间不要互相冲突。有了内存管理技术，我们就可以分别给这两个程序分配不同的物理内存，使得他们可以互不干扰。也就是说，他们认为他们访问的地址都是XXX，实际上一个程序访问的物理内存地址为YYY，另一个程序访问的物理内存地址为ZZZ。

页式内存管理技术

内存管理技术有很多中，具体可以参照理论课程。在这里，我只介绍我们实验中所采用的页式内存管理技术。

何为页式内存管理？

简单而言：我们把物理内存按照特定的大小（在我们的实验中是4KB）划分，划分出来的每一块就叫做一个"页"。

当一个程序需要使用内存的时候，我们从空闲的页中划分出若干页供其使用。当程序运行结束或者释放内存的时候，再把这些页给放回空闲页的链表中。

道理我都懂，但是我怎么具体去管理给每一个程序分配的页呢？还有上面提到的，使得每个程序拥有自己“独立”的内存空间，要怎么做？

一级页表

这个时候我们引入一个结构，叫做**页表**

先介绍较为简单的一级页表，我们实验中实际用的**二级页表**后面再介绍。

对于32为MIPS而言，其虚拟内存范围为0~4G，而我们把4K做为一个页表，则会把这4G分为2^{20}个页表。我们一般采用32位的数据，即一个int，来存储一个页表的**信息**（比如该页的物理内存地址是多少，是否有效）。那么我们需要4MB的内存空间去存储。

页表实际上就是上文中提到的4MB的内存空间。他类似于一个内存的“地图”，或者是一个微缩的内存。

注意：页表中的每一项存储的是对应页的**信息**，而不是这一个页本身。

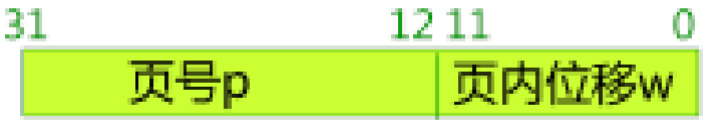
形象一点，我们可以认为页表是一个int数组：

```
const int SIZE = 1<<20;
int pgdir[SIZE];//注意，内部存的是页的相关信息。而不是页本身。
```

如果熟悉verilog的同学，可以认为此时内存是这样构造的：

```
parameter SIZE = 1<<20;//1M
parameter PAGESIZE = 1<<12;//4K
reg [SIZE-1 : 0] mem[PAGESIZE-1 : 0];
```

当我们按4KB为一页，对内存进行划分的时候，我们实际上一个32位的地址就成了下面的结构（4K=（1<<12））：



也就是说，假设有一个虚拟地址`vaddr`，那么只需要`vpn=vaddr>>12`即可得到这一个地址对应的页在页目录中的编号vpn，称为**虚页号**。

那么直接访问`pgdir[vpn]`即可获取该页的信息，比如说这一页的内存是否有效之类的。

具体而言，`pgdir[vpn]`的结构是这样子的

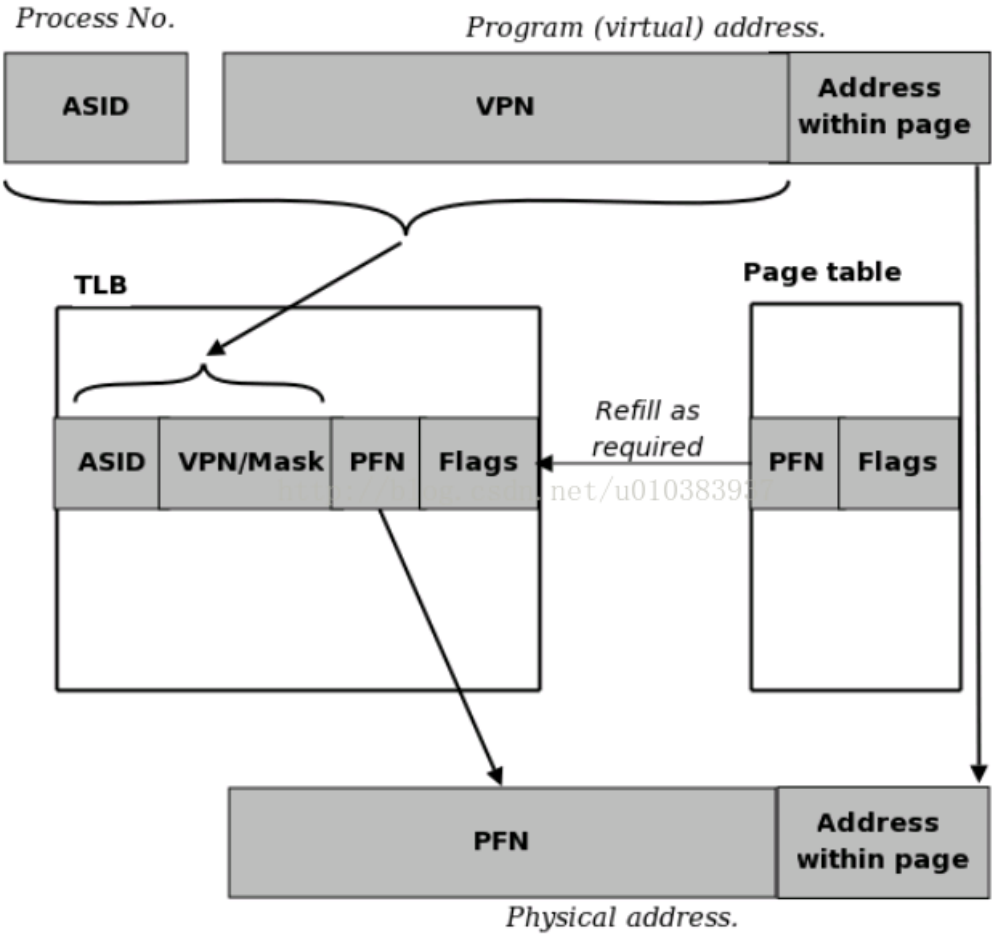
高20位	低12位
物理页编号(ppn)	标志位

因为每一个物理页是4KB，至多有$4G/4K=2^{20}$个物理页。所以想找到物理页的入口，只需要用20位的二进制去编码即可。而剩下的12位，可以用来当作标志位，比如是否有效，是否可写。具体可以参考`include/mmu.h`这个文件的35行。

MMU的转换过程

操作系统运行的时候，每一个进程会分配一个**ASID**用于区分不同的进程。

接着，对于一个虚拟地址而言，把**虚页号**和**ASID**发给MMU，MMU查找**TLB（快表，可以看作是MMU的缓存）**中是否存在该项，如果存在，直接返回物理页号。如果不存在，则启动回填程序，回填完毕之后，返回物理页号。



二级页表

一级页表已经能解决很多问题了，但是一级页表的设计中，需要4MB的内存去存储一份页表。也就是说，所有程序至少占用4MB的内存，着实有点浪费。

页表本质是一个对于内存“按需分配”的管理结构，那么我们可以以此类推，如果我们把页表也“按需分配”呢？

这样我们就得到了二级页表。

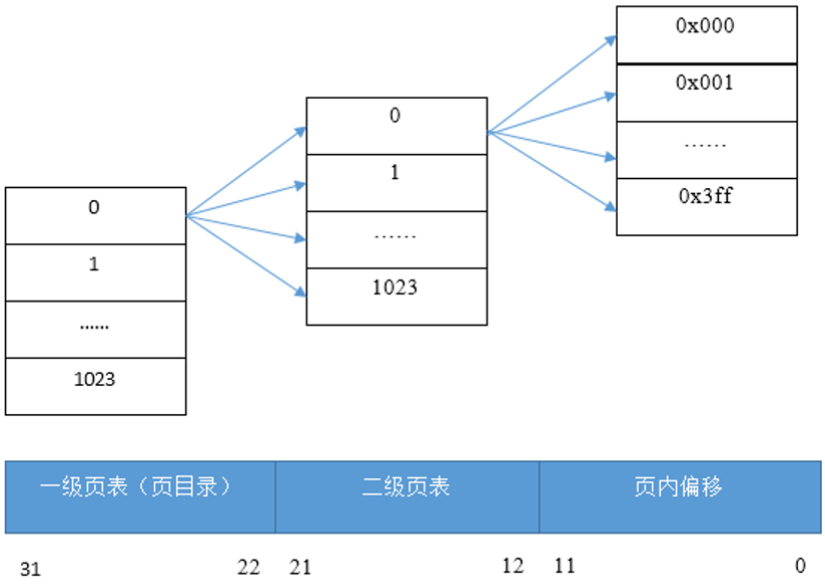
我们有：$4MB = 1024 * 4KB$

那么我们只需要1024项的页目录，即$1024 * 4B = 4KB$的空间就可以把我们的页表给再通过一个页表去存储下来，而这刚好是一页的空间。

所以我们现在就有两级页表：

- 第一级页表：
  - 又称页目录
  - 有1024项，占用大小为4KB
  - 每一项对应着一页，即4KB的内存地址。这4KB的内存地址里面可以存放1024项二级页表。
- 二级页表
  - 按需分配，以4KB，即1024项的单位去分配
  - 每一项对应着一页的物理内存地址，即4KB
- 空间计算：
  - 一级页表能管理4MB的内存空间，这内存空间是用来放二级页表的
  - 4MB的二级页表，总共能有2^{20}项，每项管理的空间是4KB
  - 所以总共能管理4G的内存空间
- 自映射

- 有一个原理是：一张世界地图，扔在世界的任意一个角落，地图上总是有且仅有一个点表示的位置正好是其落在地球上的位置
- 同样的道理，一级页表总会被映射到自身的某一项对应的二级页表中。
- 因此二级页表总共占用的内存不是4MB+4KB,而是4MB



此时，如果用verilog书写，我们可以认为内存是这么构造的：

```
parameter SIZE = 1<<10;//1K
parameter PAGE_PER_PTE = 1<<10//1K
parameter PAGESIZE = 1<<12;//4K
reg [SIZE-1 : 0] mem[PAGE_PER_PTE-1 : 0][PAGESIZE-1 : 0];
```

关于页表的一些补充说明

无论是一级页表（页目录），还是二级页表。其内部存储的信息都是如下所示：

高20位	低12位
物理页编号	标志位

要访问其对应项的时候，一般是先清空低12位，得到如下结果：

高20位	低12位
物理页编号	000

这就是对应内容的物理地址。

再把物理地址转换为内核虚地址，便可访问其内容。

开始实验

Exercise 2.1

背景说明

之前介绍了，如何管理虚页的方法，即页表。

但是对于物理页而言，可能一张物理页同时被几个进程所引用，或者是同一个进程中的不同虚页所引用。所以我们还需要对物理页进行管理。

我们采用的是空闲链表的方式，即把所有未分配的物理页串成一串，同时分别记录被分配的物理页的被引用次数等信息。

因此，第一步是把链表相关的内容给进行完善。

实验内容

> 在阅读 queue.h 文件之后，相信你对宏函数的巧妙之处有了更深的体 会。请完成 queue.h 中的LIST_INSERT_AFTER函数和LIST_INSERT_TAIL函数。

实验提示

- 利用搜索引擎，思考为什么宏定义函数要写成do……while的形式
- 这一个头文件中定义的链表，是以一个“附加模块”的形式去添加到自定义的结构体中的，仔细阅读，理解其使用方法
- 思考`LIST_ENTRY中`，`struct type **le_prev`的作用，以及为什么要这么定义

Exercise2.2

背景说明

在`init/init.c`中的`mips_init`函数会对操作系统进行初始化，这也是后面我们经常要用到的函数。

他依次调用了`mips_detect_memory();`,`mips_vm_init();`,`page_init();`这几个函数对操作系统进行初始化。

初始化内存管理的第一步，是确定可用的内存大小等信息。在正式的操作系统中，这些信息应当由BIOS提供，但是在我们的实验系统中，是由我们手动指定的大小。

实验内容

> Exercise2.2 我们需要在mips_detect_memory()函数中初始化这几个全局变量，以 确定内核可用的物理内存的大小和范围。根据代码注释中的提示，完成mips_detect_memory() 函数

实验提示

- 我们的物理内存为64MB，而一页的大小为4KB。

关于alloc函数

在系统刚刚启动时，页式内存管理还没有完成初始化。此时我们需要手动地去分配内存。

仔细阅读`mm/pmap.c`中的`alloc`函数，观察其是怎么分配物理内存的。分配内存的方向是向上还是向下?

阅读后你应该会发现，所谓的分配物理内存，实际上只是分配了一个“数字”。更深一步的内存管理细节，请继续阅读相关代码。

Exercise 2.3

背景说明

在调用了`mips_vm_init()`函数之后，我们给内核的**页目录**，物理页表管理数组**pages**，进程管理数组**envs**(在后面的lab才会用到这个东西)分别分配了内存，并且给映射到了对应的虚拟地址上。

其实，映射到虚拟内存所使用的函数目前**还没有完善**，不过一开始就去写这个函数可能难度较大。我们先从后面比较简单的函数开始写起。

假设我们现在已经完成了虚拟内存的分配工作，我们现在得记录下来**物理页**的引用情况，并且把没用过的物理页加入空闲链表中。此后我们的内存分配，均以页作为单位进行了。

实验内容

> Exercise 2.3 完成 page_init 函数，使用 include/queue.h 中定义的宏函数将未分配 的物理页加入到空闲链表 page_free_list 中去。思考如何区分已分配的内存块和未 分配的内存块，并注意内核可用的物理内存上限。

实验提示

- pages数组，与物理页表是线性映射的关系，也就是说pages[0]代表的是第一张物理页，即$[0, 4KB)$的这一部分内存
- 在此前的`alloc()`函数中，内存是向上分配的。也就是说低于`freemem`的内存现在均已被使用，引用次数应该被标记为1。（注意，freemem是虚拟地址，应先转换成物理地址再计算）
- 而剩下的内存空间，全部是没有使用的，引用次数应该被标记为0，并且加入到空闲链表中
- 思考如何在物理内存地址与物理页号之间进行转换

Exercise 2.4

背景说明

空闲页表链建立好之后，要想使用页式管理，还需要完成申请页表以及释放页表的两个操作。

page_alloc():

- 从空闲链表中获取一个空闲页
- 计算这一页对应的物理页的虚拟地址，并清空这一页中的数据

page_free():

- 如果该页的引用次数大于0，则什么都不做
- 如果该页引用次数等于0，则把这一页放回空闲页表链中
- 如果该页引用次数小于0，报错

实验内容

> 完成 mm/pmap.c 中的 page_alloc 和 page_free 函数，基于空闲内 存链表 page_free_list ，以页为单位进行物理内存的管理。并在 init/init.c 的函数 mips_init 中注释掉 page_check();。此时运行结果如下。

```
main.c: main is start ...
init.c: mips_init() is called
Physical memory: 65536K available, base = 65536K, extended = 0K
to memory 80401000 for struct page directory.
to memory 80431000 for struct Pages.
pmap.c:  mips vm init success
*temp is 1000
phycial_memory_manage_check() succeeded
panic at init.c:17: ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
```

实验提示

- 这两个函数中，均**不会改变页表的引用次数**

- 可以查看`include/queue.h`，看看链表相关的函数怎么使用
- 要掌握如何从pages转换成虚拟地址的方法，如果遇到困难，可以逐层查看`page2kva()`这个函数。

  (位于`include/pmap.h`)

Exercise 2.5

背景说明

现在我们该把之前的虚拟地址到物理地址的转换等功能给补上了，否则我们只是在"假装"管理内存而已。

此时需要我们完成两个函数：`boot_pgdir_walk`，`pgdir_walk()`

这两个函数返回某一虚拟地址在某一页目录中对应的二级页表入口地址。并且根据参数决定，如果对应的这一片二级页表是缺失的，是否新建一页。

区别：

boot_pgdir_walk

- 这个函数是在页表管理相关结构建立之前被调用的，因此无法用空闲链表等方法分配内存
- 必须手动用alloc进行内存的分配
- 由于此时pages数组还没有初始化，无需修改ref

pgdir_walk

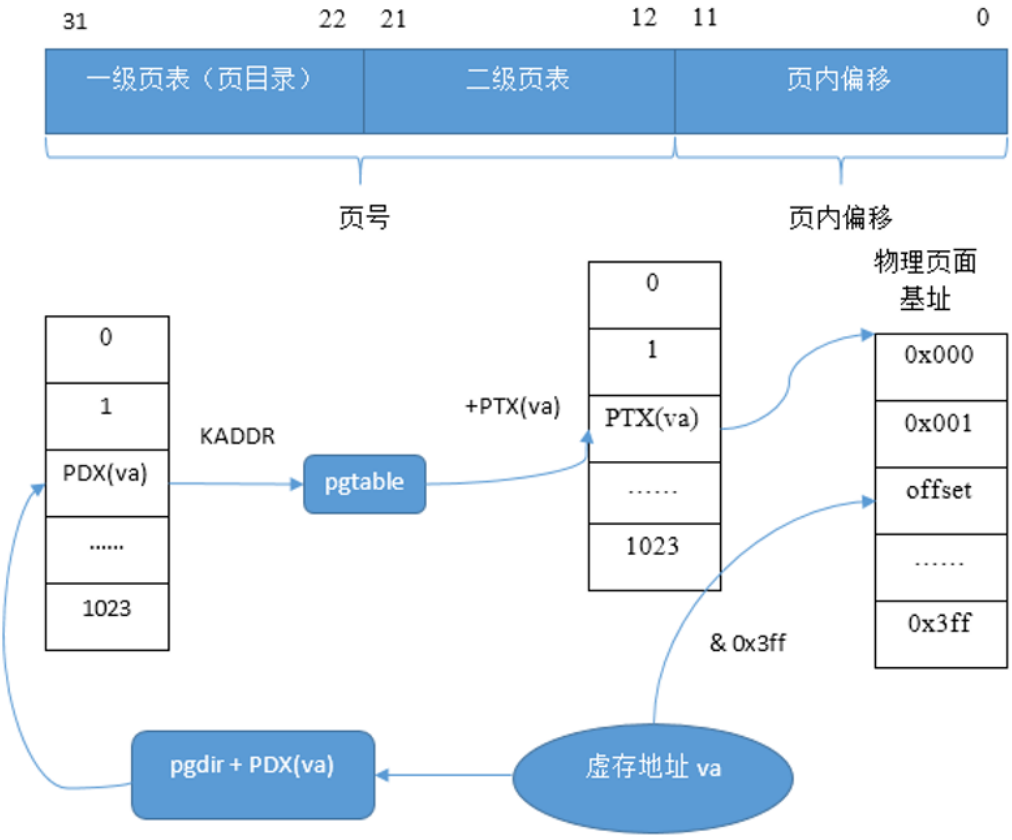
- 这个函数调用时，已经可以用页式内存管理，所以此时采用的是`page_alloc`来进行内存分配
- 如果新建了一页，需要把对应页的引用次数加1

实验内容

> Exercise 2.5 完成 mm/pmap.c 中的 boot_pgdir_walk 和 pgdir_walk 函数，实现 虚拟地址到物理地址的转换以及创建页表的功能。

实验提示

- 一定要记住，需要访问某一片内存时，要通过**虚拟地址**去访问
- 页表中填写的内容，是**物理地址**与上**标志位**
- 新建一页时，记得把标志位设置为有效
- 下图中的"物理页面基址"就是所谓的`pgtable_entryp`，也就是我们需要返回的值



Exercise 2.6

背景介绍

我们现在已经完成了提供一个虚拟地址，在指定的页目录中查找二级页表入口的功能。

但是光是能找到这个入口并没有什么用，我们还需要建立物理内存与虚拟地址之间建立映射的函数。

我们此时需要完成的函数是 `boot_map_segment`,其作用是在pgdir对应的页表上，把 `[va,va+size)` 这一部分的虚拟地址空间给映射到 `[pa,pa+size)` 这一部分空间。

实验内容

> Exercise 2.6 实现 mm/pmap.c 中的 boot_map_segment 函数，实现将制定的物理 内存与虚拟内存建立起映射的功能。

实验提示

- 顾名思义，这个函数是在boot阶段执行的，页式管理此时还没能使用，因此，请使用 `boot_pgdir_walk()` 去寻找页表入口
- 映射的内存区间可能大于一页，所以应当逐页地进行映射
- 注意要设置对应的标志位

Exercise 2.7

背景介绍

我们已经完成了系统启动时候的内存映射，页表建立等工作。现在我们需要完成一个函数以方便其他程序进行内存映射。从现在开始，我们终于不用手动分配内存了，而是采用页式内存管理系统。

在内部还会调用一个汇编函数，叫做`tlb_invalidate`,这个函数后面我们再补充实现。

实验内容

> Exercise 2.7 完成 mm/pmap.c 中的 page_insert 函数

实验提示

- 大致流程如下：
  - 先纯查找页表入口（即二级页表缺失的时候不新建）
  - 如果发现对应的页表项已经映射了一个物理页：
    - 如果这一个物理页是我们需要插入的，那么修改标志位，更新tlb。函数直接返回。
    - 如果这个物理页是其他的页，则先去除这一页。
  - 更新tlb
  - 以创建模式查找页表入口，如果查找错误，则返回错误代码
  - 把待插入的物理页的物理地址，以及标志位给填入页表项中。
  - 增加页表引用计数
- 可能用到的函数有：
  - `pgdir_walk`
  - `pa2page`
  - `page_remove`
  - `tlb_invalidate`
  - `page2pa`

Exercise 2.8

背景介绍

在上一个练习中，我们用到了`tlb_invalidate`这个函数，这个函数的作用是使得某一虚拟地址对应的tlb表项**失效**。从而下次访问这个地址的时候诱发tlb重填，以保证数据最新。

具体这个函数是怎么实现的呢?

首先，函数会先查找tlb中，对应这个虚拟地址的表项，如果没查找到，那么还原现场之后直接返回。

如果查找到了，把这一个表项更换成将`0x0`这个虚拟地址映射到`0x0`物理地址的一条映射关系。而0号内存地址是永远不会被用到的，也就是说增加了一个无效的记录。

也就是说，用一个永远不会用到的映射关系去"挤掉"之前的记录。以保证下次在tlb查找的时候，找不到当前记录。

实验内容

> Exercise 2.8 完成 mm/tlb_asm.S 中 tlb_out 函数。

实验提示

MIPS寄存器的介绍：https://www.mips.com/?do-download=the-mips32-and-micromips32-privileged-resource-architecture-v6-02

MIPS指令的介绍：https://www.mips.com/?do-download=the-mips32-instruction-set-v6-06

重点查看tlbwi和tlbp这两个指令，ENTRYHI,ENTRYLO0,ENTRYLO1这三个寄存器。