OS Lab2 - ExprimentReport

1. 思考题

1. 请思考cache用虚拟地址来查询的可能性，并且给出这种方式对访存带来的好处和坏处。另外，你能否能根据前一个问题的解答来得出用物理地址来查询的优势?

既然虚拟地址和物理地址一样，都是用来定位数据的，故理论上来说是可行的。

好处:

使用物理地址来查询，则会有MMU介入CPU和cache之间，造成读写数据的性能的损失，而使用虚拟地址，则可以将CPU给出的地址（虚拟的）直接到cache中去寻找，使得速度得到提高。

坏处:

1.不同的进程有时有相同的逻辑地址对应不同的物理地址的情况，或者多个逻辑地址对应同一个物理地址的情况，若不用其他方法这种情况则会在切换进程时发生错误。

2.若cache没命中，需要重新通过MMU获得物理地址到内存中查找。

2.破坏各个用户、各个进程之间的独立性，造成安全隐患。

物理地址查询的优势:

使用物理地址，则CPU的逻辑寻址必然经过一次地址转化（无论是清空高位的kseg0, kseg1，还是需要经过MMU转换的kuseg, kseg2）为物理地址，不会出现进程切换时因为地址映射问题而发生的错误。也保证了各个用户、进程的安全性和独立性。且使得对于CPU而言，外部设备（unmapped）与cache的寻址是统一一致的。

2. 在我们的实验中，有许多对虚拟地址或者物理地址操作的宏函数(详见include/mmu.h),那么我们在调用这些宏的时候需要弄清楚需要操作的地址是物理地址还是虚拟地址，阅读下面的代码，指出x是一个物理地址还是虚拟地址。

```
int x;
char* value = return_a_pointer();
*value = 10;
x = (int) value;
```

虚拟地址。一般程序中的指针均指向虚拟地址空间。

3. 我们在 include/queue.h 中定义了一系列的宏函数来简化对链表的操作。实际上，我们在 include/queue.h 文件中定义的链表和 glibc 相关源码较为相似，这一链表设计也应用于 Linux 系统中 (sys/queue.h 文件)。请阅读这些宏函数的代码，说说它们的原理和巧妙之处。

原理即数据结构中各种链表的插入、删除、遍历等方法。巧妙之处如下：

1. 定义的结构具有很好的扩展性，在实际使用时，可以通过在某一结构体中定义`LIST_ENTRY(结构体名称) 链接的指针的总的名称(field)`使得这一结构体变成可以被链接的对象。
2. 对于各种方法，实现了较好的封装，可以想见其他程序想要调用这些结构是十分方便的
3. for_each循环宏让我们很好地对链表的元素进行了遍历（自己从来没想过还有这种方法所以强调下~）

4. 我们注意到我们把宏函数的函数体写成了 do { /* ... */ } while(0)的形式，而不是仅仅写成形如 { /* ... */ } 的语句块，这样的写法好处是什么?

使得宏函数的使用更不容易出错。我们C语言编程的习惯要在每行结尾处增加`;`，对于前一种宏函数正确，而对于后一种将会出现编译错误。

5.注意，我们定义的 Page 结构体只是一个信息的载体，它只代表了相应物理内存页的信息，它本身并不是物理内存页。 那我们的物理内存页究竟在哪呢？Page 结构体又是通过怎样的方式找到它代表的物理内存页的地址呢？ 请你阅读 include/pmap.h 与 mm/pmap.c 中相关代码，并思考一下。

物理内存页存储在内存中，其地址由如下方式确定：

在pmap.h中定义了如下函数

```
static inline u_long
page2pa(struct Page *pp)
{
    return page2ppn(pp) << PGSHIFT;
}

static inline u_long
page2ppn(struct Page *pp)
{
    return pp - pages;
}
```

其中pages是存储所有Page*的全局大数组，pages本身代表其首地址。

当然也有其逆过程

```
#define PPN(va)     (((u_long)(va))>>12)
static inline struct Page *
pa2page(u_long pa)
{
    if (PPN(pa) >= npage) {
        panic("pa2page called with invalid pa: %x", pa);
    }

    return &pages[PPN(pa)];
}
```

而kva和pa之间的转换关系由宏函数定义

```
#define ULIM 0x80000000
define PADDR(kva)                       \
    ({                                  \
        u_long a = (u_long) (kva);              \
        if (a < ULIM)                   \
            panic("PADDR called with invalid kva %08lx", a);\
        a - ULIM;                       \
    })

#define KADDR(pa)                       \
    ({                                  \
        u_long ppn = PPN(pa);                   \
        if (ppn >= npage)                   \
            panic("KADDR called with invalid pa %08lx", (u_long)pa);\
        (pa) + ULIM;                    \
    })
```

6.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(请注意指针)。

C，下面给出正确的格式：

```
struct Page_list{

    struct {

         struct {

            struct Page *le_next;

            struct Page **le_prev;

        } pp_link;

        u_short pp_ref;

    }* lh_first;

}
```

7. 在 mmu.h 中定义了 bzero(void *b, size_t) 这样一个函数,请你思考，此处的b指针是一个物理地址， 还是一个虚拟地址呢?

虚拟地址。因为在pmap.c中，需要分配时清零该块空间时，调用了bzero函数，CPU实际对某个页面进行清0操作时，都是用的虚拟地址。

8. 了解了二级页表页目录自映射的原理之后，我们知道，Win2k内核的虚存管理也是采用了二级页表的形式，其页表所占的 4M 空间对应的虚存起始地址为 0xC0000000，那么，它的页目录的起始地址是多少呢?

0xC0300000

9. 注意到页表在进程地址空间中连续存放，并线性映射到整个地址空间，思考：是否可以由虚拟地址直接得到对应页表项的虚拟地址？上一节末尾所述转换过程中，第一步查页目录有必要吗，为什么?

不能，二级页表机制本身要求：一级页表即页目录的每一项存储的是页表项的物理地址，经过转换后才成为对应页表项的虚拟地址，不能直接由虚拟地址获得对应页表项的虚拟地址。因此有必要。

10. 观察给出的代码可以发现，page_insert 会默认为页面设置PTE_V的权限。请问，你认为是否应该将PTE_R 也作为默认权限？并说明理由。

不应该作为默认权限，在mmu.h中定义`PTE_R`如下：

```
#define PTE_R       0x0400  // Dirty bit ,'0' means only read ,otherwise make
interrupt
```

可见PTE_R不同于PTE_V， PTE_V在插入后可以保证存储着正确的物理地址，即该页表项是有效的；而PTE_R是对该页面的权限位，当设置为0时为只读模式，设置为1时才可以向该页面写入内容，这并不是在page_insert函数调用的时候能够确定的。因此不应该将其作为默认权限。

11. 思考一下tlb_out 汇编函数，结合代码阐述一下跳转到NOFOUND的流程？从MIPS手册中查找tlbp和tlbwi指令，明确其用途，并解释为何第10行处指令后有4条nop指令。

1.将CP0_ENTRYHI的原值存入k1寄存器，并将需要查找的页表项的虚拟地址传入CP0_ENTRYHI；

2.tlbp在tlb中查找是否存在该虚拟地址，如果存在则找到该项，并将该页表项清零；若未找到该项，则跳转到NOFOUND，恢复CP0_ENTRYHI为原值。

3.函数返回。

此函数的作用即为：传入一个页表项的虚拟地址，若该地址在TLB中存在，则将该项清0，否则保持TLB不变。

`TLBP`(TLB_Probe)：如果TLB中不存在待查询的页表项，则index的最高位被置1；反之若找到待查询的页表项，则返回该页表项在TLB中的位置。

`TLBWI`(TLB_Write_Index)：向TLB[Index]中写入Entry页表项，即对TLB[Index]进行更新。

tlbp需要多个时钟周期才能完成，增加nop指令使CPU空转，确保该指令执行完成后再准确读取index的值，防止出现错误。

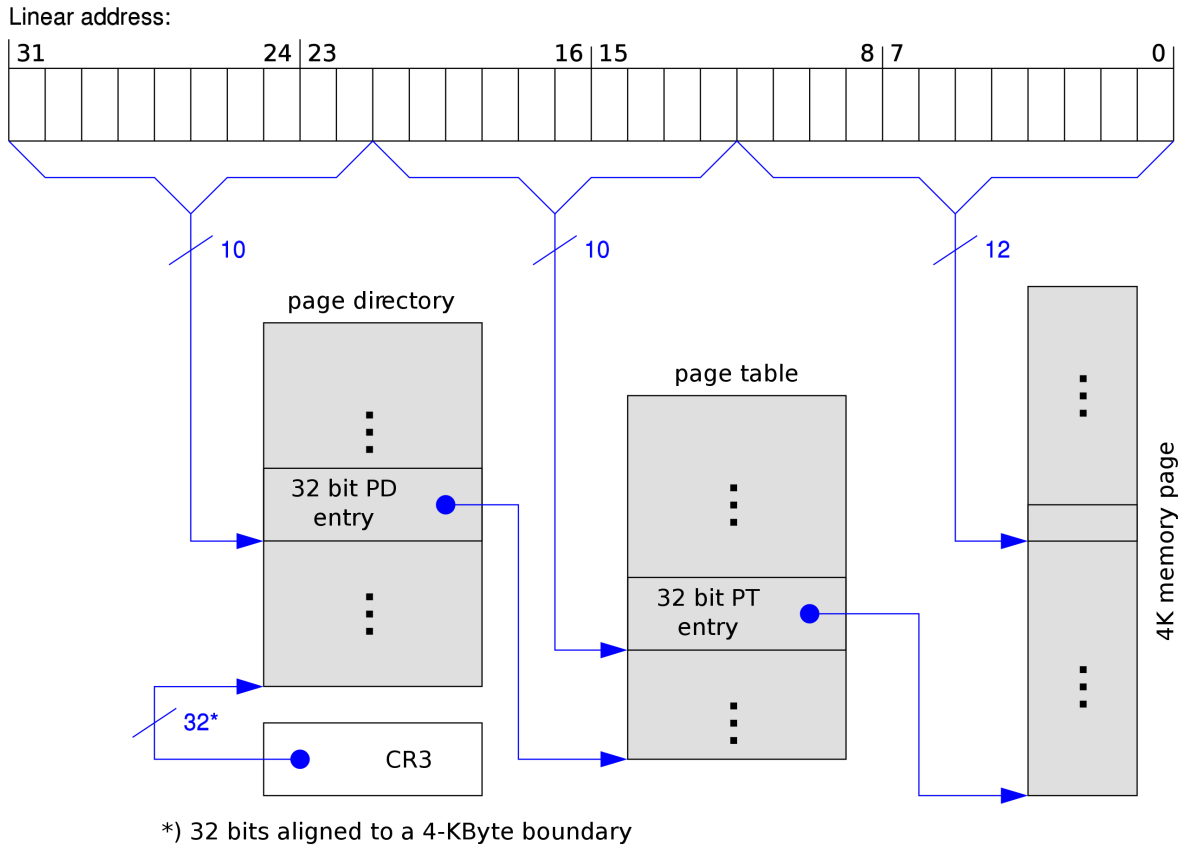
12. 显然，运行后结果与我们预期的不符，va值为0x88888，相应的pa中的值为0。这说明我们的代码中存在问题，请你仔细思考我们的访存模型，指出问题所在。

我们本次的TLB实现不完整，缺少缺页异常时的处理机制（重填机制等），这样，当我们调用`page_insert`函数中的`TLB_out`汇编函数时，由于没有找到时没给出相应的缺页的处理机制，故该页表项始终没有被加入TLB中，导致向其中写入值时，始终无法写入且陷入死循环。因此pa中的值没有被改变。

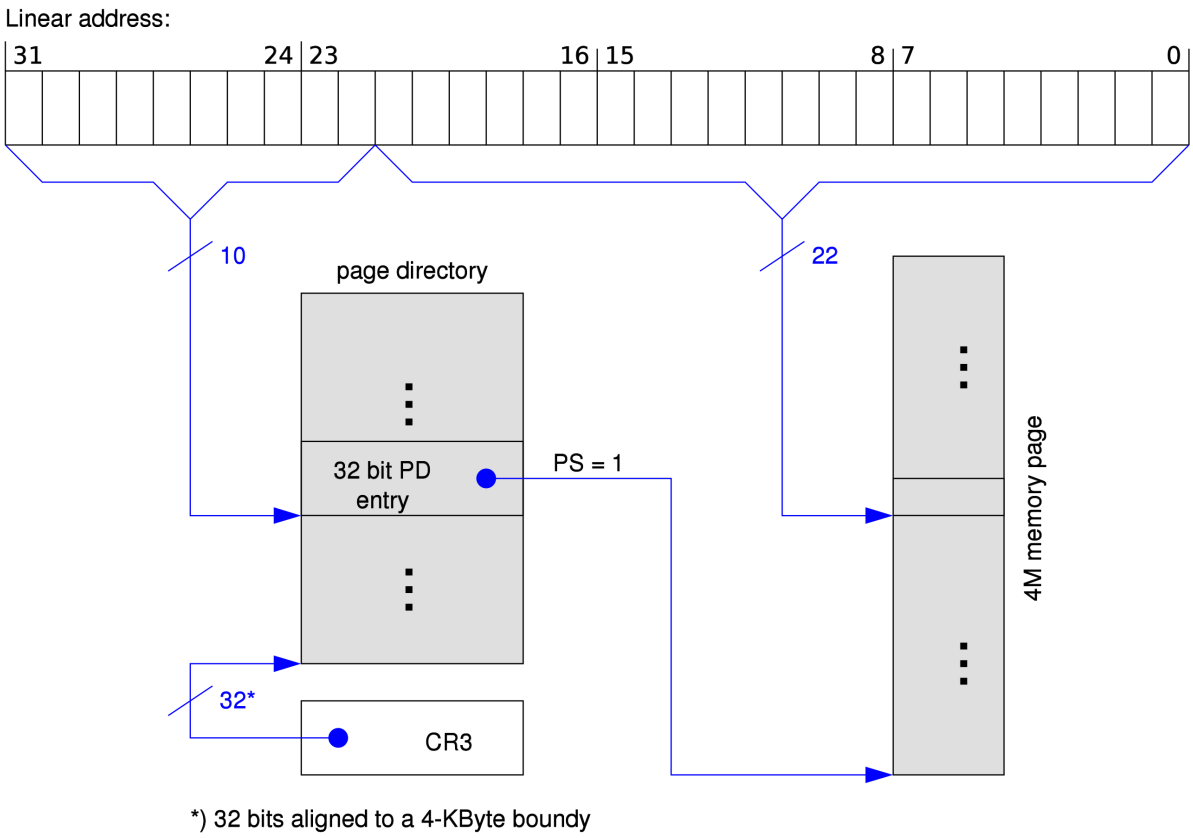
13. 在X86体系结构下的操作系统，有一个特殊的寄存器CR4，在其中有一个PSE位，当该位设为1时将开启4MB大物理页面模式，请查阅相关资料，说明当PSE开启时的页表组织形式与我们当前的页表组织形式的区别。

PSE（Page Size Extension）：如果CR4的PSE = 1的话，这个页目录表项就指向了一个4MB的大物理页面，而不是象原来那样指向一个页表。同时，使用PSE技术的页目录项实际 上只使用了传统IA32页目录结构中20个地址位的高10位（后面的32 - 10 = 22位在寻址的时候全部填0），所以，它指向的是一个4MB对齐的物理页面.下图来源于wikipedia：

普通模式下的二级页表结构：

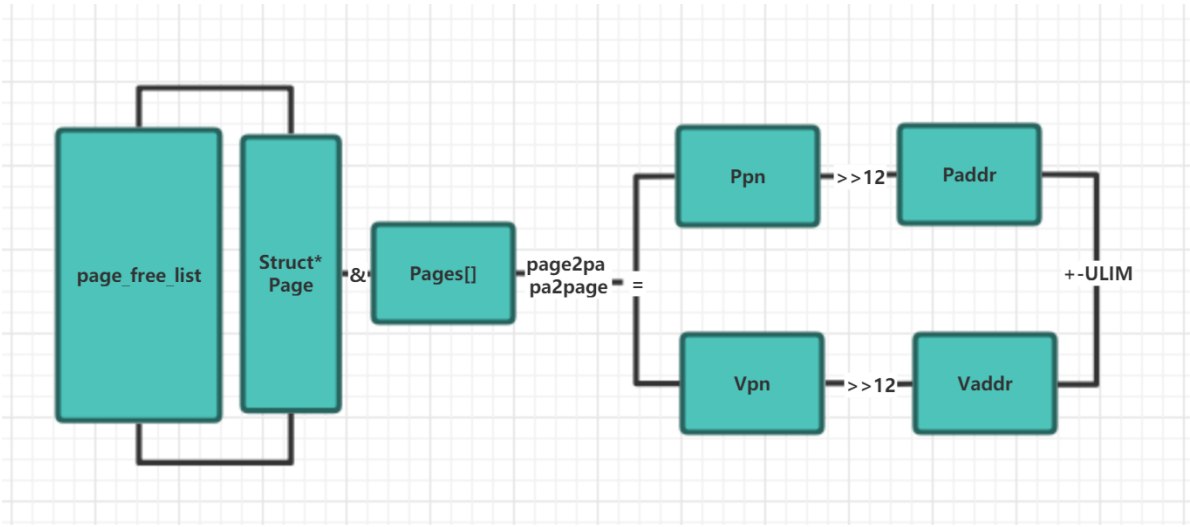


PSE的page后22位全部为页内偏移，故只需要用前十位在page_directory中寻找巨页的地址，如下图所示：



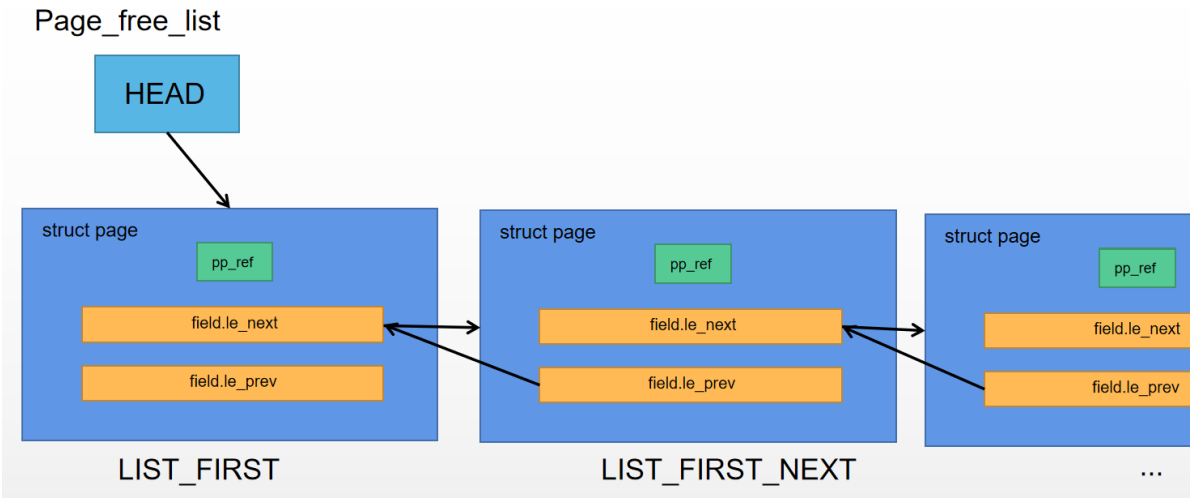
2.实验难点图示

1.地址、page、page_queue间的相关关系：

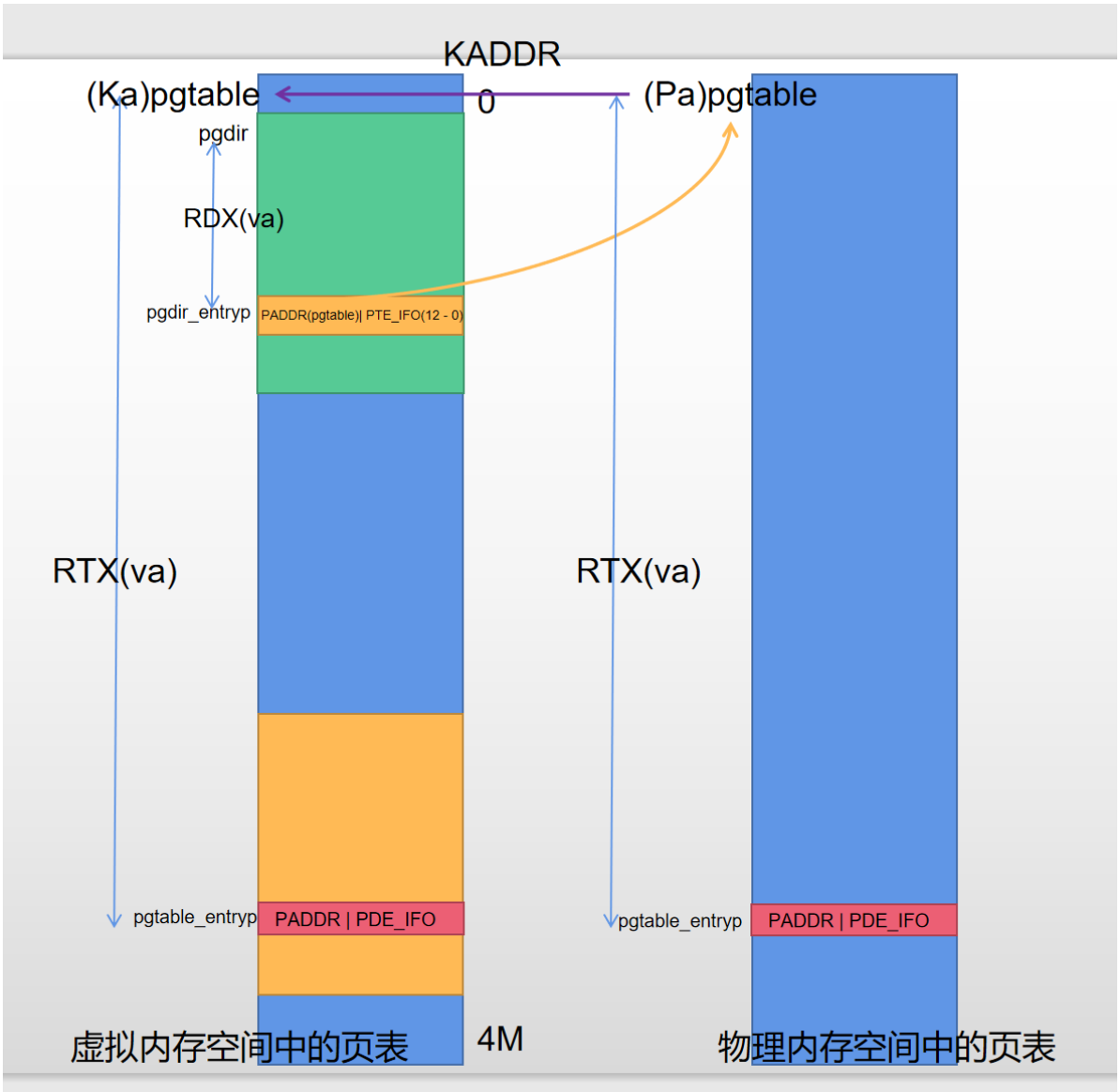


注意：这里的VADDR与PADDR可以直接通过+-ULIM实现是因为该部分代码属于操作系统的内核代码，页表本身存储在Unmapped的内核空间中，无需通过页表机制进行转化。而页表本身记录的数据则用于用户实际进程的PADDR与VADDR的转化。

2.page_free_list：

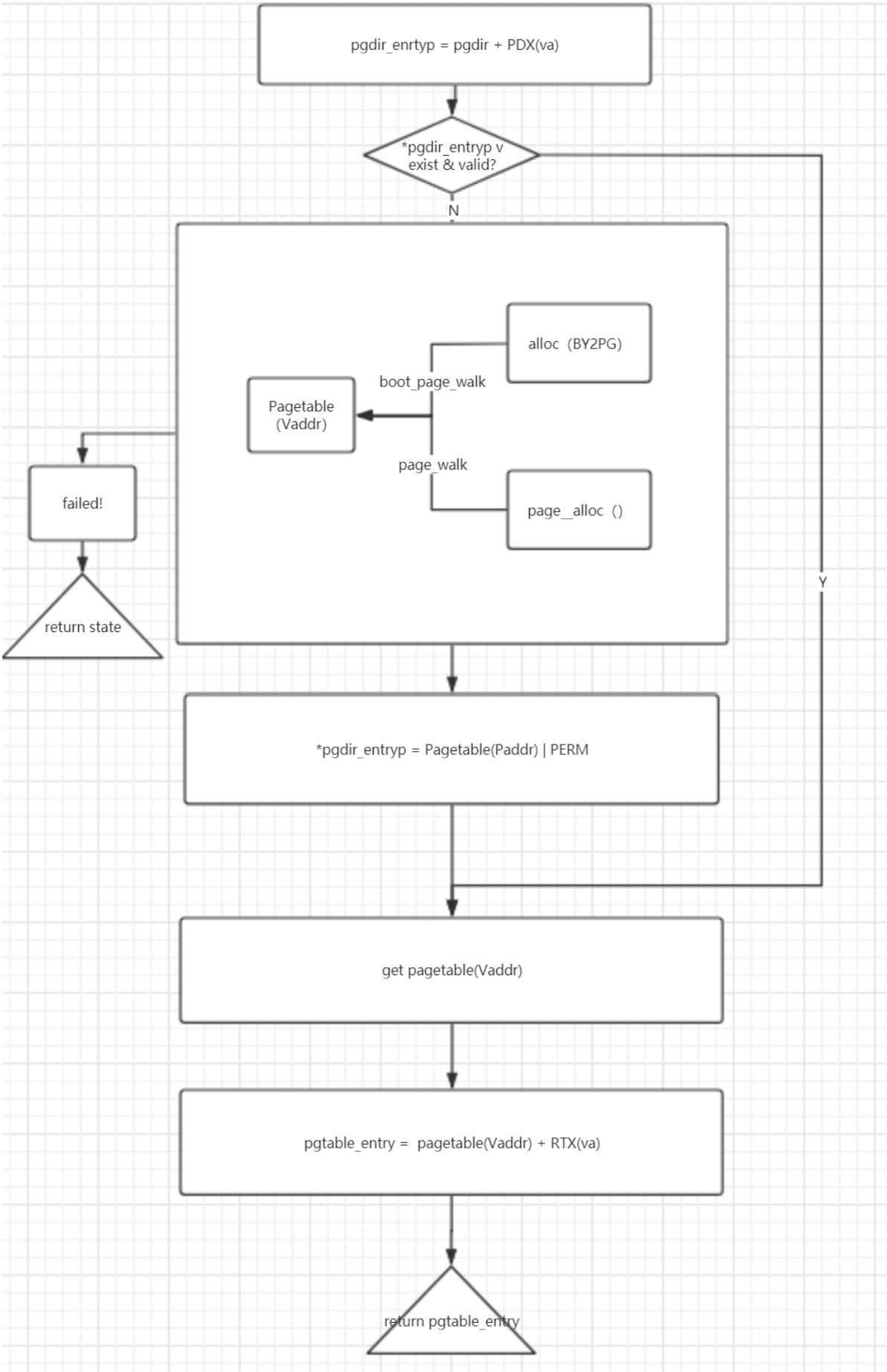


3. 自映射二级页表结构及 page_walk函数的实现手段（通过va找到该va对应的页表项的入口地址）



注意：由于页表本身存储在无需经过MMU映射的区域，故可以通过KADDR宏实现物理地址向虚拟地址的转换，再基于pagetable（VADDR）加上页表索引数（RTX），即可找到该va对应的的页表项的入口地址，其中记录着应当被翻译的物理地址（或需要记录）。

4. (boot_)page_walk函数的实现（通过va找到该va对应的页表项的入口地址）



3.实验总结

本次实验可以说是OS的一个重点实验，难度也是较大的。也因此花费了不少时间。个人认为本次实验的核心理解在于区分什么时候使用虚拟地址，什么时候使用物理地址。最后可以感觉到：由于pmap.c本身是运行的程序，我们**进行任何的地址的操作和访问时使用的都是虚拟地址**，**物理地址仅作为数据存储于页表中或者仅作为转化的中间过程。**

对于本次实验的流程：课下的作业部分重点考察了物理内存的管理方式（包括未建立页表机制前按需分配内存的函数、建立页表机制后使用链表结构维护空闲页表等）；虚拟内存中二级页表的映射。课上实验部分考察了对页表所处的占用、空闲等状态与其数据结构本身的联系（Lab2-1-exam），构建新的数据结构取代链表维护页表（Lab2-1-Exra）；对二级页表自映射结构的理解（Lab2-2-exam），统计每个物理页表项被映射的次数（Lab2-2-Exra）。但最后lab2的两个Extra都白给了，究其原因是自己对内存管理机制的理解还有所欠缺。当然这也给了我一些教训：如果想要挑战Extra，必须要细致阅读代码并做到较好的理解，并能够进行灵活运用。

4.指导书反馈

1. 代码中函数命名的问题：

   首先，希望能够给出每个取单词部分缩写的函数名称的全名，感觉有时候真的不知道全程是啥有点让自己摸不着头脑。

   其次，对于两个函数的名称，感觉取的不太恰当，具体如下：

   page_insert函数的作用是给定一个页表ppage和va，在页表中将va对应的页表项中存储的物理地址设置为为该给定页表ppage的物理地址。从这一角度理解，改名称为page_update更为合适。

   page_remove函数的作用是给定一个va，在页表中将va对应的页表项中存储的内容清零。从这一角度理解，改名称为page_clear更为合适。

2. 感觉缺少一些让我们更加理解实验的图（其实某种程度上是自己懒得探索总结~）

5.残留难点

对于自映射页表的理解还是不是很到位。比如较难理解这段代码：

```
pgdir_entryp = pgdir + PDX(va);
pgtable_entry = pgtable + PTX(va);
```

见难点图示2，这里对pgtable_entry使用 PTX（va>>12）作为pgtable_entry的偏移量不是很能理解，如果像图示中的那样强行理解，则lab2_2_Extra中遍历二级页表时的方法就没有办法解释了。