1. **思考题**

**Thinking 2.1  
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?**

**是可能的，可在原本的索引条件下加上虚页号索引。**

**优点：**

**从逻辑上看简单，流水线利用虚拟PC访问cache，而cache直接支持虚拟地址访问，不需要转换机构，实现起来较直接。**

**缺点：**

**（1）若两段进程的虚拟PC相同，会出现虚拟地址访问cache冲突的问题，需要处理。**

**（2）操作系统依赖页表中的保护机制确保自己不被改写，cache若直接用虚拟地址访问，就绕过了页表机制，使得指令执行不存在保护。**

**（3）现代操作系统允许多个虚拟页面空间映射到同一个物理地址页面空间，若纯粹是虚拟地址访问，可能出现读写不一致的情况，需要处理这个问题，存在性能损失。**

**（4）外部IO设备地址映射大多是物理地址映射，CPU若直接读写外部设备，TLB可以将虚拟地址映射为物理地址，但没有机制将物理地址映射为虚拟地址。**

**使用物理地址查询的优势：**

1. **进入cache的物理地址是唯一的，不会出现cache访问冲突；**
2. **页表中的额保护机制可以发挥作用，确保操作系统自身的安全；**
3. **在TLB中处理了多个虚拟页面空间映射到同一个物理空间的问题；**
4. **能够解决读写外部IO设备的问题。**

**Thinking 2.2  
请查阅相关资料，针对我们提出的疑问，给出一个上述流程的优化版本，新的版本需要有更快的访存效率。（提示：考虑并行执行某些步骤）**

虚拟地址访问cache，物理地址比较Tag

1. CPU给出虚拟地址来访问数据，TLB接收到这个地址之后查找是否有对应的页表项，进行虚实地址转换，同时利用虚拟地址的低位访问cache，得到cache存储的物理Tag与data信息。
2. 若TLB与cache均命中，则直接返回给CPU数据
3. 若TLB命中，cache缺失，则MMU执行正常的页表查询工作之后再根据物理地址在cache中查询，同时更新TLB中的内容。
4. 若TLB缺失，cache命中，则需要通过Tag比较cache中的data信息是否就是虚拟地址要访问的，若是，则直接返回给CPU数据，并更新TLB，若不是，则调用MMU机制查询物理地址，在cache中查询，并更新TLB，返回数据。
5. 若二者均缺失，则同样需要MMU执行页表查询物理地址，并按照相应算法进行cache的替换或装填，之后返回给CPU数据。

**Thinking 2.3**

**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。**

X是一个虚拟地址。

**Thinking 2.4  
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？**

结构更清晰，可有效分隔函数体与宏定义的代码，do{ }while(0)仅将花括号内的语句执行一次。

**Thinking 2.5  
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，给出你的想法。**

Maxpa开辟了64M的物理内存，这些物理内存空间以4KB为一页分为了0x4000(npage)页，因此我们的物理内存全部都被按页划分，物理内存页分布于整个物理内存空间。

通过在pmap.h的page2pa得到page代表的物理内存页的地址。

计算这个Page相对于结构体数组起始位置的偏移，乘以物理页的大小，再加上物理内存基址(也就是0)，就能找到这个物理页的起始地址。

**Thinking 2.6  
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

**Thinking 2.7  
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。在pmap.c以及与之相关的C语言函数中，使用\*的基本上都是虚拟内存。C语言函数无法之间对物理地址进行操作。

**Thinking 2.8  
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？**

(0xC0000000>>12)\*4 + 0xC0000000 = 0xC0300000

**Thinking 2.9  
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。**

**用途：**

tlbp是TLB查询指令，在TLB中查找虚页号和ASID（地址空间标识符）跟EntryHi寄存器中当前内容相匹配的表项，并把相应表项的索引值存入Index寄存器。若匹配失败，Index寄存器的位31被置1，这使得它看起来像个负数，很容易检测。

tlbwi根据Index寄存器写TLB表项，用来在Index寄存器所选中的表项和EntryHi与EntryLo0-1寄存器之间传递MMU的相关数据。

Index寄存器用来选择特定的TLB表项，使用tlbp对TLB表项进行软件搜索后，Index寄存器会自动设置。

**流程：**

先将EntryHi寄存器中的内容取出放在k1寄存器中，再将a0寄存器中的值，也就是传入tlb\_out的PTE\_ADDR(va)值存入EntryHi寄存器，然后用tlbp在TLB中查找虚页号和ASID（地址空间标识符）跟EntryHi寄存器中当前内容相匹配的表项，并把相应表项的索引值存入Index寄存器。

用mfco取出Index寄存器中的值到k0寄存器，使用”bltz k0,NOFOUND”指令，若第一步中tlbp匹配失败，则k0寄存器中32位为1，小于0，跳转到NOFOUND；若第一步中tlbp匹配成功，则bltz跳转条件不成立，程序继续向下执行。

可以注意到最后均需要执行

mtc0 k1,CP0\_ENTRYHI

j ra

nop

语句，是由于读取TLB表项时，会覆盖EntryHi(ASID)域，操作系统会继续选择当前正在运行的程序的地址空间，所以需要用”mtc0 k1,CP0\_ENTRYHI”将EntryHi(ASID)写回原值。

**4条 nop指令原因：**

tlbp指令对CP0\_INDEX寄存器进行了写入操作，而紧接着下一条指令需要读到这个寄存器，对于流水线CPU，这个操作可能需要4个时钟周期，分别比较EntryHi, EntryLo0, EntryLo1, PageMask这四个寄存器的值，所以需要塞入4个nop阻塞，来保证读到的CP0\_INDEX寄存器为最新值。

**Thinking 2.10  
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。**

pa = va2pa(boot\_pgdir, va);这条指令查询到va对应的物理地址pa，而pa对应地址超出了Kernel Virtual Address的范围。

**Thinking 2.11  
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。**

我们的小操作系统中开辟的每一个页表大小为4KB，开启大物理页面模式后，每一个页表da小变为da4MB，也可以看作是1024个4KB的页表聚合在一起，直接使用页目录号去寻找物理页框号对应的页表基地址。

1. **实验难点**
2. 通过二级页表进行虚拟地址到物理地址的转换

1.通过PDX(va)获得一个虚拟地址对应的页目录索引

2.凭借索引在页目录中得到对应的二级页表的基址(物理地址)

3.通过(KADDR)将物理地址转化为内核虚拟地址

4.通过PTX(va)获得这个虚存地址对应的页表索引

5.从页表中得到对应的页面的物理地址

1. 创建页表

mm/pmap.c boot\_pgdir\_walk

pgdir\_walk

PDX获得页目录索引(31-22)

PTE\_ADDR 得到物理页框号(31-12)

PTE\_V 标志位

PADDR(pgtable)|PTE\_V 置位于低12位的标志位

PTX 获得对应页表索引(21-12)

1. 地址映射

mm/pmap.c boot\_map\_segment 将指定的物理内存与虚拟内存建立起映射

boot\_map\_segment(pgdir, UPAGES, n, PADDR(pages), PTE\_R);

其中

（1）pgdir为上面pgdir = alloc(BY2PG, BY2PG, 1);为其分配的地址，80401000，

是内核地址

（2）UPAGES 为mmu.h中定义的页表地址，为0x7f80 0000

（3）n为 npage\*sizeof（struct Page）

（4）PADDR（pages） 是页对应的物理地址，pages同样是内核地址

由(struct Page \*)alloc(npage \* sizeof(struct Page), BY2PG, 1)给出

（5）PTE\_R是标志位，或一下就好了

4.TLB汇编函数

我参考《SEE MIPS RUN LINUX》第6章中TLB/MMU的控制指令一节，大概了解了tlbp与tlbwi指令。

1. **体会与感想**

Lab2的实验对大批量代码的阅读要求进一步提高，我们需要从指导书的部分描述以及不断揣摩各段代码的目的来理解整个内存管理的流程。实验指导书采用了先物理内存后虚拟内存的顺序，先实现对物理内存的管理，接着建立二级页表，接着建立页表与物理页的映射，最后处理TLB的步骤，带领我们一步步深入内存管理。我觉得Lab2-1的实验难度不高，但是Lab2-2就开始有点晦涩难懂。需要阅读一段代码多次或者到别处查阅资料来实现对实验原理的理解。

在物理内存与虚拟内存映射方面，我觉得我个人理解的还不是很透彻，对于boot\_map\_segment函数中的每一句话，我都需要进一步揣摩它的意图。page\_insert同样需要施以更多关注。

1. **指导书建议**

关于Page结构体、物理页以及虚拟内存之间的联系，指导书能否给出一个更清晰的解答。我自己在琢磨这三者之间的关系时感觉特别痛苦。