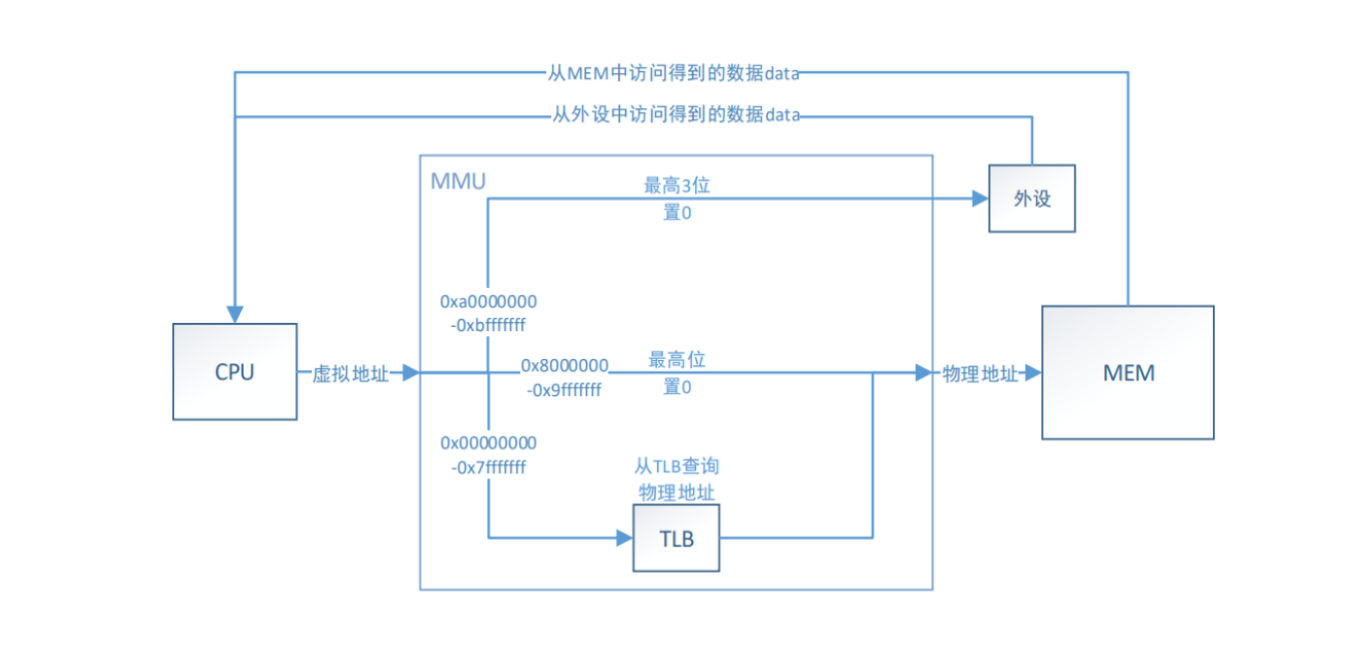
**操作系统LAB2实验报告**

**210617 21373037 杨博文**

1. **思考题**

**1.在编写的C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 和 sw 使用的是虚拟地址，还是物理地址？**

代码在CPU内执行时得到的都是虚拟地址。



**2. 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。**

用宏实现链表，主要是可以模拟出C语言不具有的泛型功能。因为宏的实现方式是字符串替换，替换时不会考虑到语法等等问题，因此只要给好参数，则任何结构体的链表均能由一组宏统一实现，简单便捷。

**插入操作(考虑先寻找，再插入)**

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)

双向循环链表: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)

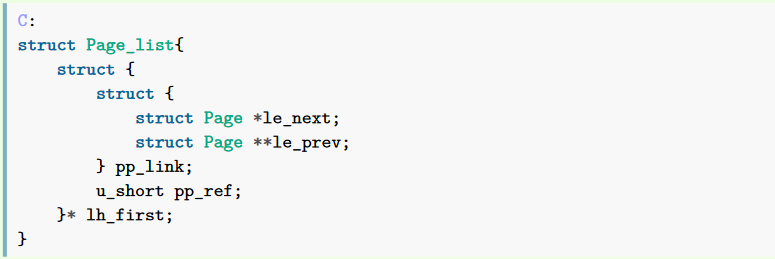
**删除操作:**

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)

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

**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。**



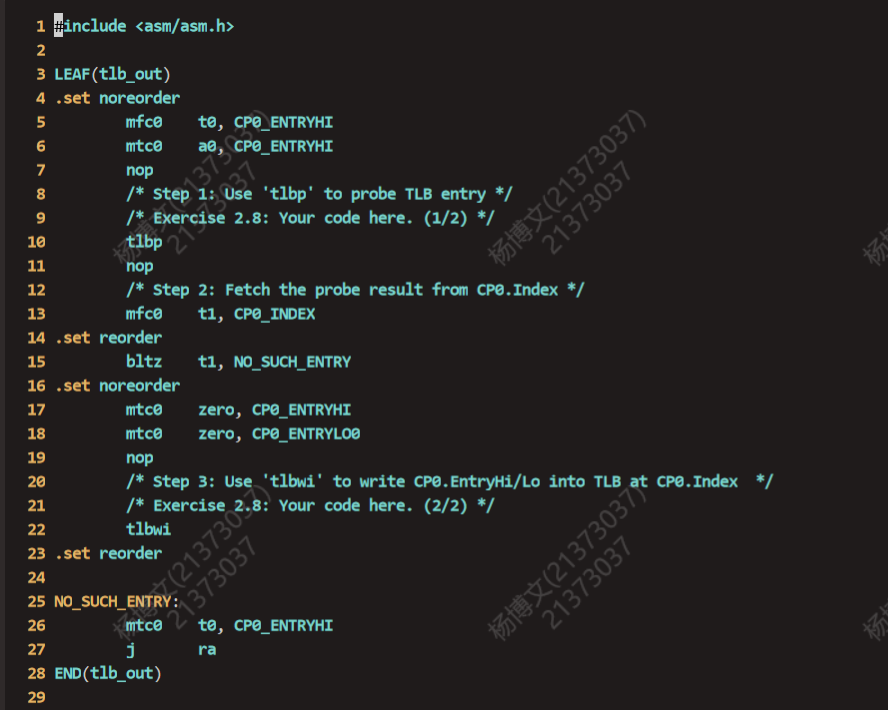
选C，Page\_list是Page结构体的头指针，里面包含一个指向Page结构体的指针即lh\_first，Page结构体由引用次数pp\_ref和链表项pp\_link组成，故选C。

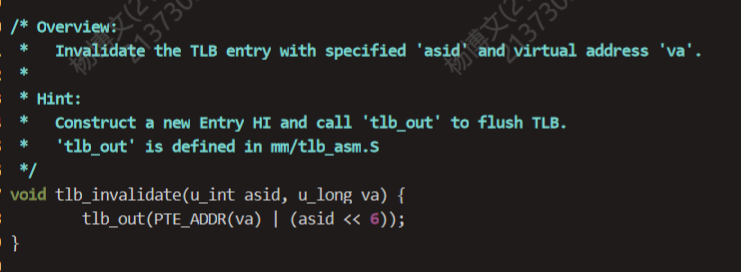
**4.** **请阅读上面有关 R3000-TLB 的描述，从虚拟内存的实现角度，阐述 ASID 的必要性。请阅读《IDT R30xx Family Software Reference Manual》的 Chapter 6，结合 ASID段的位数，说明 R3000 中可容纳不同的地址空间的最大数量。**

ASID称作是地址空间，每个用户进程有一个专属的ASID码，用来区分不同进程。由于每个进程都有自己的虚拟地址空间且相同，在MIPS上都是0-4G,因此进程间会使用到相同的虚拟地址，但实际上他们对应的是不同的物理地址，若此刻不增加标志位来区分不同进程，TLB便分不出来不同的进程，出现严重错误，当然也可以每切换一次进程刷新所有TLB重新装填，但是这样效率会大大降低。

ASID共六位，一共最多容纳64个地址空间。

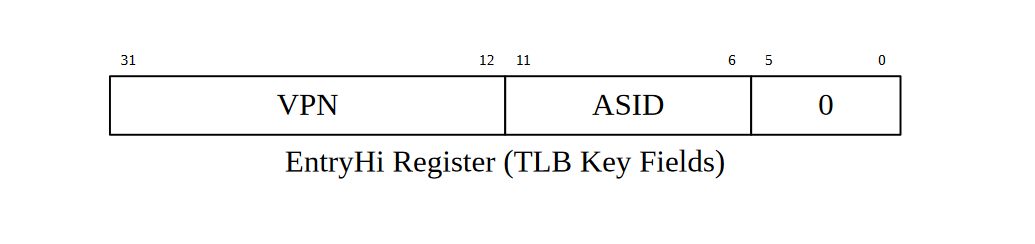
**5. tlb\_invalidate 和 tlb\_out 的调用关系？请用一句话概括 tlb\_invalidate 的作用。逐行解释 tlb\_out 中的汇编代码。**

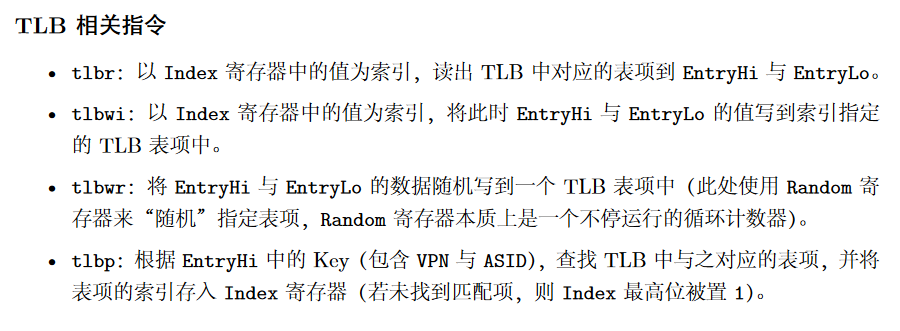
 **tlb\_out代码如下:**

**tlb\_invalidate代码如下:**

tlb\_out 被 tlb\_invalidate 调用，tlb\_invalidate的作用是删除特定(即特定ASID码)虚拟地址在 TLB 中的旧表项。

PTE\_ADDR作用是清空va虚拟地址后十二位偏移位，接着与位移到6到11位的ASID码进行拼接，生成 TLB的Key，如下图。





结合相关指令,按行解读tlb\_out代码：

**mfc0 t0, CP0\_ENTRYHI**：将原EntryHi寄存器内容取出

**mtc0 a0, CP0\_ENTRYHI**：将tlb\_invalidate向其传递的参数a0(这一步在编译器种执行)写进EntryHi寄存器

**tlbp**：根据EntryHi中的Key查找对应的Value，并将查到的表项索引存入Index寄存器

**mfc0 t1, CP0\_INDEX**：将Index寄存器值存入t1寄存器

**bltz t1, NO\_SUCH\_ENTRY**：判断t1是否为负数，若为负数，意味着无对应表项，也就不需要清空，跳转到NO\_SUCH\_ENTRY。若不为负数，意味着查到表项，继续执行。

**mtc0 zero, CP0\_ENTRYHI**：将EntryHi清空

**mtc0 zero, CP0\_ENTRYLO0**：将EntryLow清空

**tlbwi**：根据上面Index查到的索引，用清空后的Hi和Low对相应表项清空，完成表项无效化操作。

**6. 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，若三级页表的基地址为 PTbase，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）。**

首先正如二级页表结构里页目录的自映射一样，三级页表结构中，512张二级页表一定有一张正是一级页表，而每张二级页表对应的512张三级页表也一定有一张正是二级页表，因此总共有512\*512\*512=128M页表项，每个页表项对应着4KB大小的页面，共512GB,没问题。

三级映射要经历三步。PTbase对应的⻚表项是第PTbase>>12个，⽽每个⻚表有512个⻚⽬录项，每个⻚表项的⼤⼩是8B（⻚⾯⼤⼩ 4KB，有512个⻚表项），所以PTbase在三级⻚表区中对应的偏移量是PTbase>>9。记作⼆级⻚表项基地址。

然后，三级⻚表区（1G）映射到⼆级⻚表区（512\*512=256K个8B⻚表项，2M⼤⼩）。每个⻚表项映射了4K的空间，则⼆级⻚表区基地址对应的⻚表项是第(PTbase>>9)>>12个，所以⼆级⻚表区基地址在⼆级⻚表区中对应的偏移量是(PTbase>>9)>>9。记作⼀级⻚表基地址。

最后，⼆级⻚表区（2M）映射到⼀级⻚表（512个8B⻚表项，4K⼤⼩）。每个⻚表项映射了4K的空间，则⼀级⻚表基地址对应的⻚表项是第(PTbase>>18)>>12个，所以⼀级⻚表基地址在⼀级⻚表⾃⾝中对应的偏移量是(PTbase>>18)>>9。

那么，三级⻚表⻚⽬录（即⼀级⻚表）的基地址为 PTbase + (PTbase>>9) + (PTbase>>18)

映射到⻚⽬录⾃⾝的⻚⽬录项是 PTbase + (PTbase>>9) + (PTbase>>18) + (PTbase>>27)

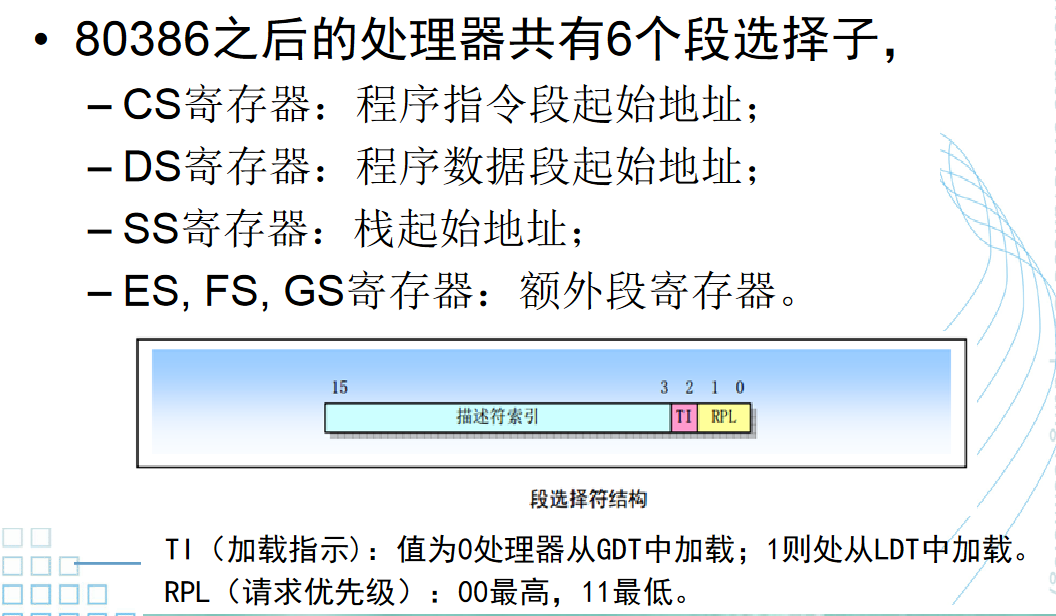
**7. 简单了解并叙述 X86 体系结构中的内存管理机制，比较 X86 和 MIPS 在内存管理上的区别。**

x86架构的内存管理机制分为两部分：分段机制和分⻚机制。 而Mips架构是单纯的分页机制。

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。分⻚机制实现了传统的按需分⻚、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，分⻚机制还可以⽤于提供多任务的隔离。X86处理器⽆论在何种运⾏模式下都不可以禁⽌分段机制，但是分⻚机制却是可选选项。

针对分段机制而言，分段相当于完成了虚拟地址到线性地址的转化，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，可以将GDT当作一级段表，LDT看作二级段表。当需要访问一个段时，需要将该段的选择子装进段寄存器中，对应的描述符从GDT或LDT取出。

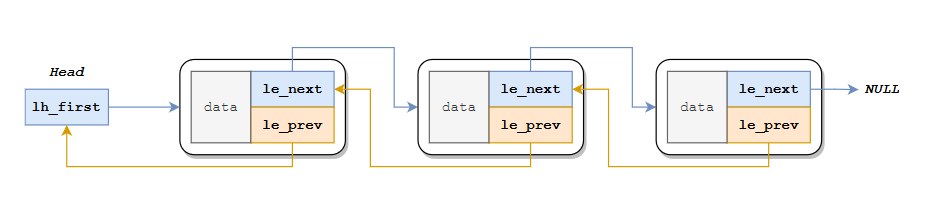
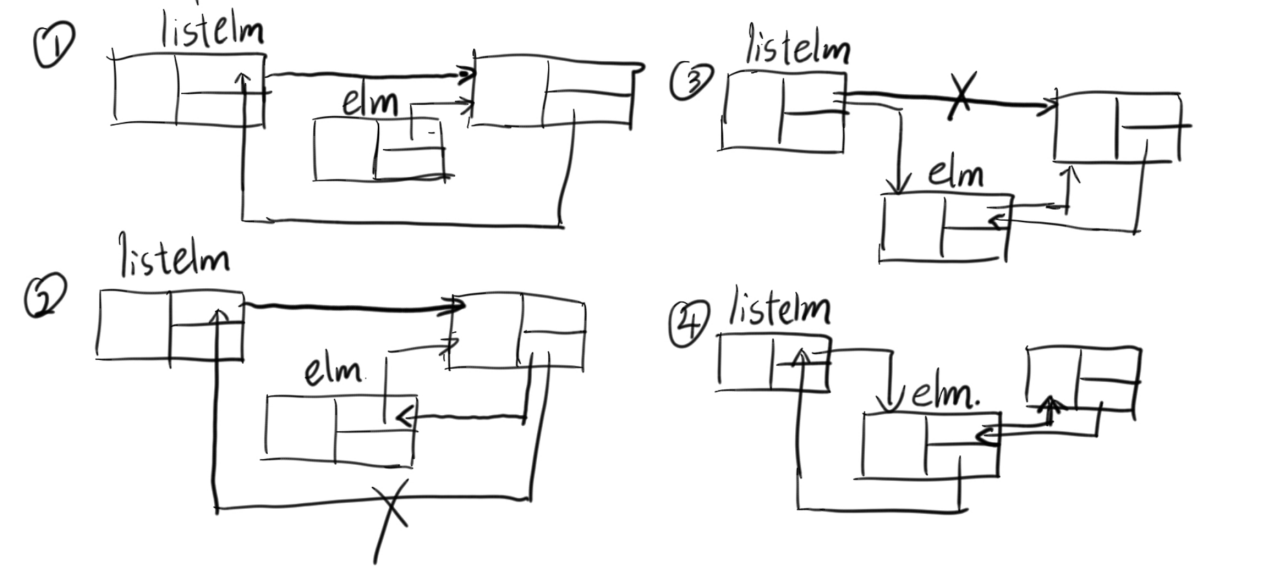
若开启分页机制，则将得到的线性地址解释成页目录偏移+二级页表偏移+页内偏移，像是纯分页机制中的“虚拟地址”继续通过页表得到实际地址。若不开启分页机制，则得到的线性地址就是物理地址。



**二.实验难点**

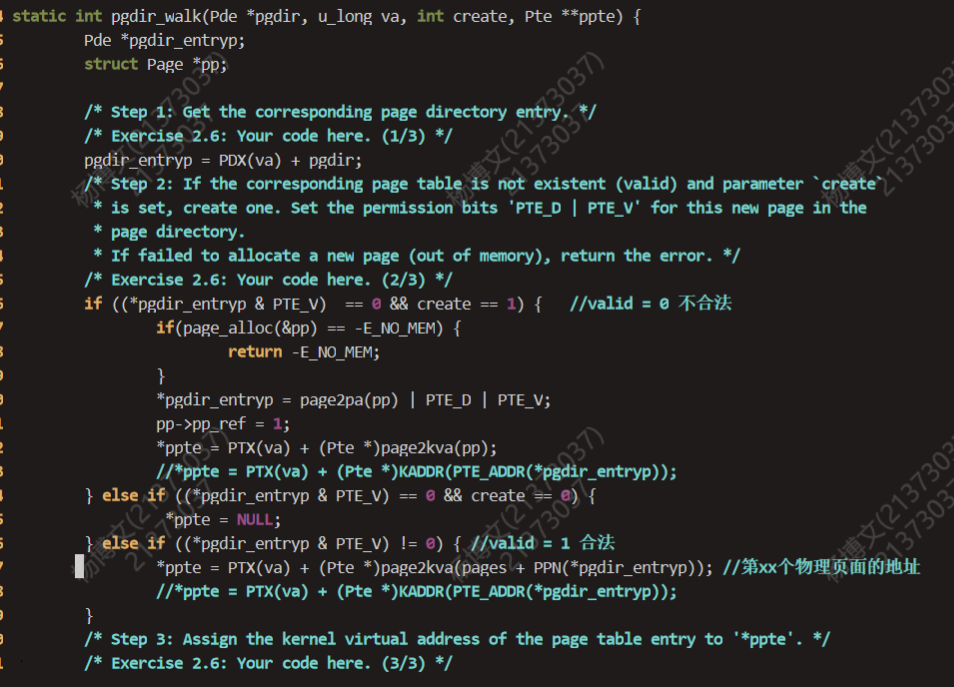
本次实验重点在于对内存管理机制的初始化工作，对物理内存分页，并对页面实现申请与删除，并完成了一些与页表相关的函数。我认为pmap.c中页表函数是本次实验的难点，接下来我将依次梳理一下各个函数的用途与实现思路，并总结这些函数搭配使用产生的最终效果。

首先是物理页面的创建与构建空闲页面双向链表，难点在于链表宏的实现，需要用到数据结构的相关知识，前插后插需要分四步，每一步的顺序至关重要，以后插为例，elm为待插入元素，listelm为当前元素，先判断listelm后是否还有元素，若无则直接尾插即可，若有则如下图四步执行(画的略难看)，特别要注意指针的指针存的是指针的地址，即le\_prev存的是le\_next的地址，应使用le\_prev = &le\_next。在构建页面时应按Page结构体物理地址从低到高与对应的物理页面地址从低到高对应。

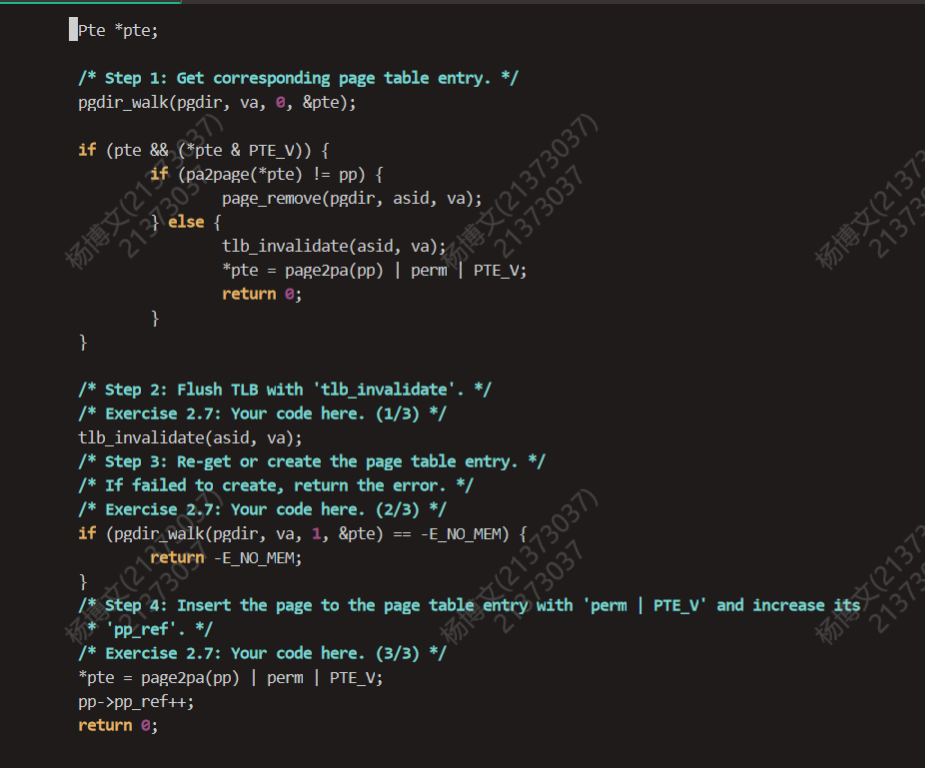
****

接下来是对几个页表函数的理解。

Page\_walk：

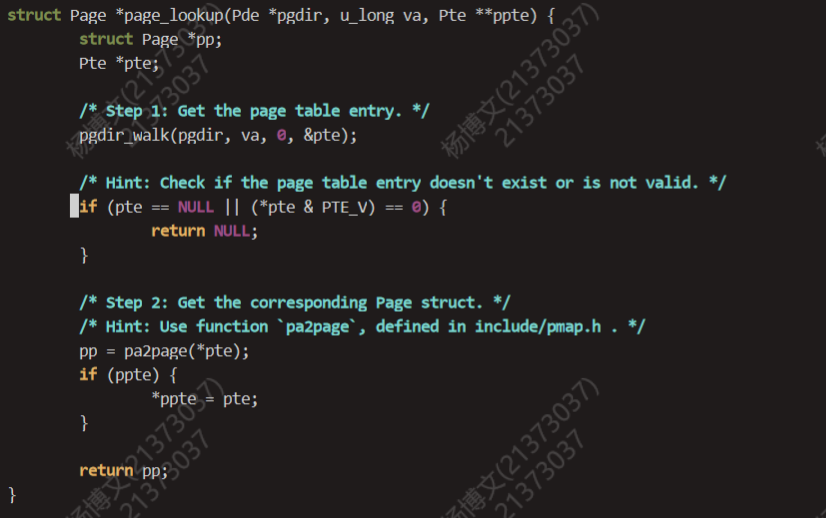
给定虚拟地址，查找页目录，若能查到相应的二级页表，将va 虚拟地址所在的二级页表项的指针存储在 ppte 指向的空间上，若没查到，则建立二级页表，完成对页表页面的初始化，即引用次数置为1，后续操作同。

Page\_insert：



将一级页表基地址 pgdir 对应的两级页表结构中虚拟地址 va 映射到页控制块 pp 对应的物理页面，若va在页表中无相应二级页表则使用page\_walk创建，若va在二级页表中对应的物理页面跟本次设置的不同，则需要清空tlb并重写二级页表，最后将对应的物理页面号填到二级页表当中，并设置权限。

Page\_look:



作用是返回一级页表基地址 pgdir 对应的两级页表结构中虚拟地址 va 映射的物理页面的页控制块，同时将ppte 指向的空间设为对应的二级页表项地址,若无则返回null，主要是用来配合tlb重填函数。

这一切的一切当发生缺页异常时，OS将执行如下步骤。

1. 从 BadVAddr 中取出引发 TLB 缺失的虚拟地址。  
   2. 从 EntryHi 的6–11 位取出当前进程的 ASID。  
   **3. 以虚拟地址和 ASID 为参数，调用 \_do\_tlb\_refill 函数。该函数是 TLB 重填过程的核心，其功能是根据虚拟地址和 ASID 使用look函数查找页表，若查找不到则会利用页面申请函数来申请一个物理页面，并使用insert函数将物理页面号存入页表之中，在insert和look的过程中又会调用walk函数查找二级页表项存不存在，直至look函数返回不为null。**4. 将物理地址存入 EntryLo , 并执行 tlbwr 将此时的 EntryHi 与 EntryLo 写入到 TLB 中。

以上函数间的调用关系便非常清晰明了，且我们发现这些页表函数统一服务于tlb重填操作，在重填的操作过程中，完成了页表的建立。

**三．实验感受**

在Lab2中，我们首次接触到了如此大型的程序，头文件C文件汇编文件应有尽有无一缺席，各种文件之间的配合也是令人眼花缭乱。在笔者拿到之初，一度无从下手，令人不禁翻看往届学长的代码，不过最后及时收手，想着还是自己先挑战挑战。最让人记忆犹新的是写page\_walk，page\_insert那个夜晚，因为create的逻辑问题，debug一度de到破防，甚至在怀疑是不是真出现了以前Lab写错的情况发生，de到在主楼沙发上撒泼打滚，好在我的朋友乐于助人，在一个个文件的来回切换中，最终锁定了问题，当时已然临近11点。好在第三部分的内容简单，Lab2实验部分在高潮之后很快便迎来尾声。不过想要理清内存管理的逻辑有些困难，想必有许多同学在写代码时只是单纯的根据提示去写，没有思考内在的逻辑，笔者也是在之后又继续翻阅指导书与代码才大致弄清逻辑。

总的来说本次实验是一次具有挑战感与收获感的实验，随着一遍遍阅读内核代码，笔者对于OS的原理也是愈发清晰，也愈发明白为什么说写OS的程序员都是最优秀的程序员了。

**Lab3冲冲冲！**