

思考题

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?

答: 指针变量中存储的地址是虚拟地址, MIPS 汇编程序中 `lw` 和 `sw` 都是虚拟地址, 因为实验使用的只会发出虚拟地址

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, 分析三者插入与删除操作上的性能差异。

答:

- 使用宏来实现链表可以使代码更加简洁和易于维护。宏定义可以将代码块作为单个实体进行操作, 从而使代码更具可读性。使用宏来定义链表节点和链表操作可以使代码更加高效, 因为宏定义的代码会在编译时被嵌入到程序中, 而不需要在运行时进行函数调用。
- 在我们的实验环境中单向列表的结构很简单进行插入操作时只需要将新插入节点的指针指向要插入的位置的下一个节点, 那么循环列表在插入和删除其实和单向列表无太大的区别。

在单向链表中, 每个节点只有一个指针指向下一个节点, 因此在插入和删除操作时, 需要找到要操作的节点的前驱节点, 然后将它的指针指向插入节点或删除节点的后继节点。这个过程需要遍历链表, 时间复杂度为 $O(n)$, 其中 n 是链表的长度。

在循环链表中, 最后一个节点的指针指向第一个节点, 因此它可以像单向链表一样进行插入和删除操作。但是, 由于循环链表的特殊结构, 它可以通过头节点或尾节点来快速定位链表的起点和终点, 从而减少遍历链表的次数, 提高插入和删除操作的效率。

在双向链表中, 每个节点有两个指针, 一个指向前驱节点, 一个指向后继节点。这样, 在插入和删除操作时, 可以直接访问前驱节点和后继节点, 不需要像单向链表那样遍历链表。因此, 双向链表的插入和删除操作的时间复杂度为 $O(1)$ 。

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。

```
1 A:
2 struct Page_list{
3     struct {
4         struct {
5             struct Page *le_next;
6             struct Page **le_prev;
7         }* pp_link;
8         u_short pp_ref;
9     }* lh_first;
10 }
```

```
1 B:
2 struct Page_list{
3     struct {
4         struct {
5             struct Page *le_next;
6             struct Page **le_prev;
7         } pp_link;
8         u_short pp_ref;
9     } lh_first;
10 }
```

```
1 C:
2 struct Page_list{
3     struct {
4         struct {
5             struct Page *le_next;
6             struct Page **le_prev;
7         } pp_link;
8         u_short pp_ref;
9     }* lh_first;
10 }
```

答: 正确结构是 C

Thinking 2.4 请思考下面两个问题:

- 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。

答:

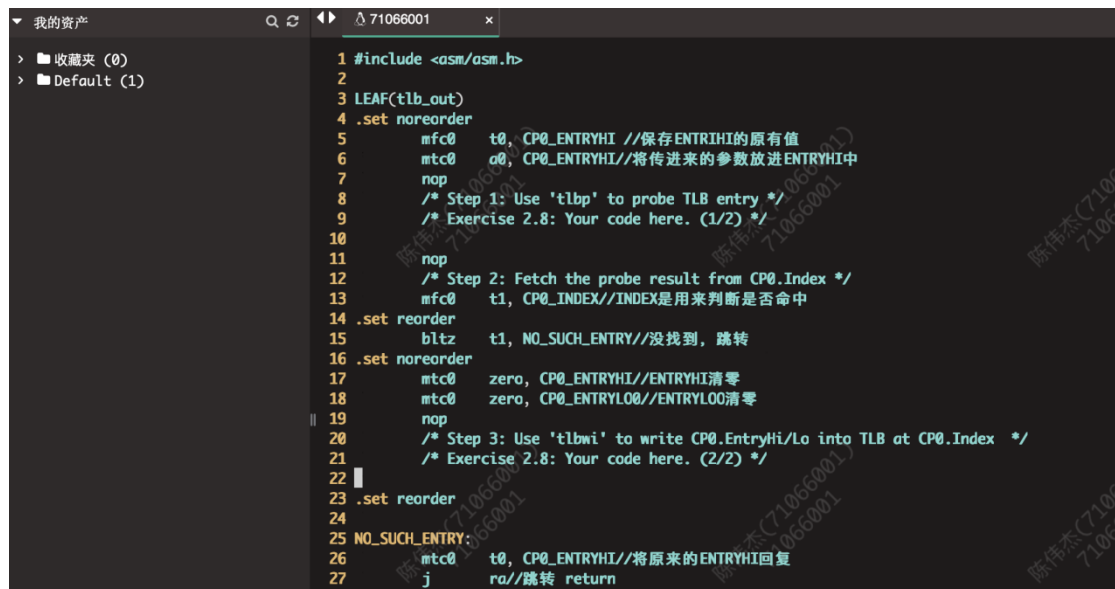
- 从虚拟的实现角度来看 ASID 可以保证不同进程的虚拟地址空间互相隔离, 避免了数据混乱和安全问题。同时还可以提高虚拟内存的访问效率, 因为 我们知道 R3000-TLB 可以缓存最近访问的虚拟地址和相应的物理地址的映射关系, 从而避免了频繁的地址转换操作。
- ASID 共 6 位 (6 bits long), 故能容纳不同地址空间的最大数量为 64 (64 address spaces)

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。

答：

- 首先 `tlb_invalidate` 和 `tlb_out` 两者的调用关系是 `tlb_out` 会在操作完成后调用 `tlb_invalidate`，以确保新映射生效。因此，在需要删除旧的地址映射并添加新地址映射时，可以连续调用 `tlb_out` 和 `tlb_invalidate`
- 一句话概括 `tlb_invalidate` 的作用就是撤销 TLB 条目，即使之前的地址映射不再有效，这样新的地址映射即可生效



```
1 #include <asm/asm.h>
2
3 LEAF(tlb_out)
4 .set noreorder
5     mfc0    t0, CP0_ENTRYHI //保存ENTRYHI的原有值
6     mtc0    a0, CP0_ENTRYHI//将传进来的参数放进ENTRYHI中
7     nop
8     /* Step 1: Use 'tlbp' to probe TLB entry */
9     /* Exercise 2.8: Your code here. (1/2) */
10
11     nop
12     /* Step 2: Fetch the probe result from CP0.Index */
13     mfc0    t1, CP0_INDEX//INDEX是用来判断是否命中
14 .set reorder
15     bltz    t1, NO_SUCH_ENTRY//没找到，跳转
16 .set noreorder
17     mtc0    zero, CP0_ENTRYHI//ENTRYHI清零
18     mtc0    zero, CP0_ENTRYLO0//ENTRYLO0清零
19     nop
20     /* Step 3: Use 'tlbwi' to write CP0.EntryHi/Lo into TLB at CP0.Index */
21     /* Exercise 2.8: Your code here. (2/2) */
22
23 .set reorder
24
25 NO_SUCH_ENTRY:
26     mtc0    t0, CP0_ENTRYHI//将原来的ENTRYHI回复
27     j       ra//跳转 return
```

Thinking 2.6 任选下述二者之一回答：

- 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。

答：X86 体系结构中的内存管理机制：

X86 采用的是基于分段和分页的内存管理机制。分段是将内存按逻辑上的需要分为若干个段，每个段有自己的基址和限长，通过段寄存器和段描述符来映射逻辑地址到线性地址；而分页是将线性地址划分为若干个页，每个页的大小通常为 4KB，通过页表将线性地址映射到物理地址。在分段分页的机制下，X86 可以实现虚拟内存管理，支持多道程序共存和内存保护等功能。

X86 和 MIPS 在内存管理上的区别：MIPS 采用的是基于分页的内存管理机制。与 X86 不同的是，MIPS 的页表是多级的，通常采用两级页表结构，即页目录和页表。页目录中的每个项指向一个页表，页表中的每个项则指向一个物理页帧，通过页目录和页表的多级映射，将虚拟地址映射到物理地址。此外，MIPS 的 TLB（快表）可以缓存常用的页表项，提高地址转换的效率。相比之下，X86 采用的分段分页机制更为复杂，但是它能够更细致地控制内存的访问权限和保护机制。

实验难点

首先在了解指针的应用（体现出有个好的 c 语言底子的重要性），两级页表结构，这次的实验很多都是在 `pmap.c` 这个文件里执行，对应的就是我们内存管理所以要理解结构体的结构

在 Exercise 2.6 里 我始终对 `pgdir_walk` 函数的理解不够深透，对实现虚拟地址到物理地址的转换以及创建页表的功能，在什么时候需要用虚拟地址，什么时候需要用物理地址,如果需要则创建一个新的页表并分配物理内存，同时在页目录中指向该页表等。我们要知道在这个函数是通过页目录直接访问页面表项，而非遍历整个页面表，也是因为自己对题目的不清晰导致在这道题消耗时间比较长。

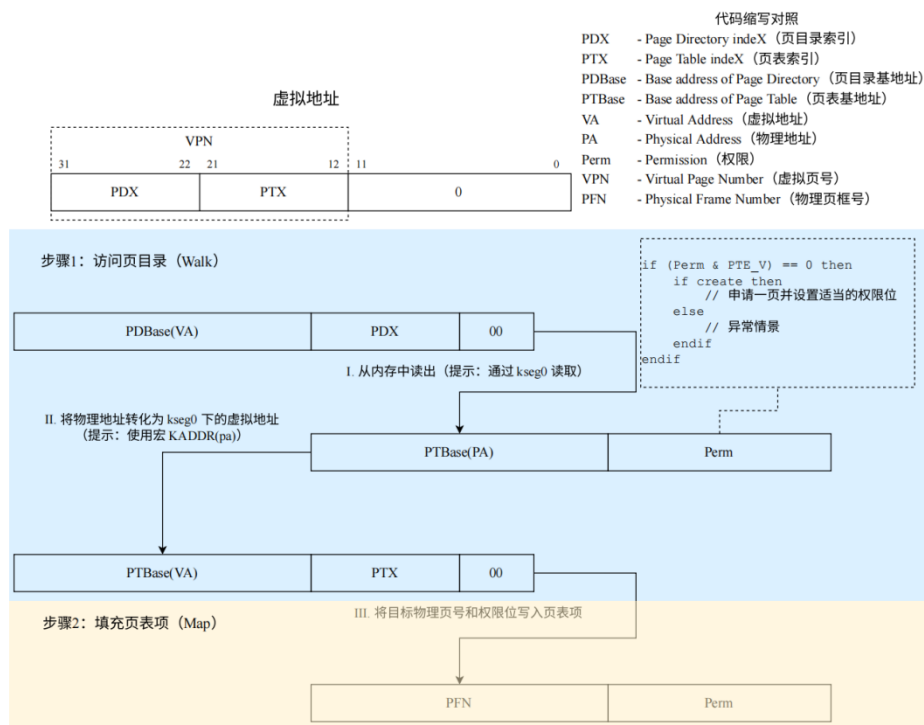
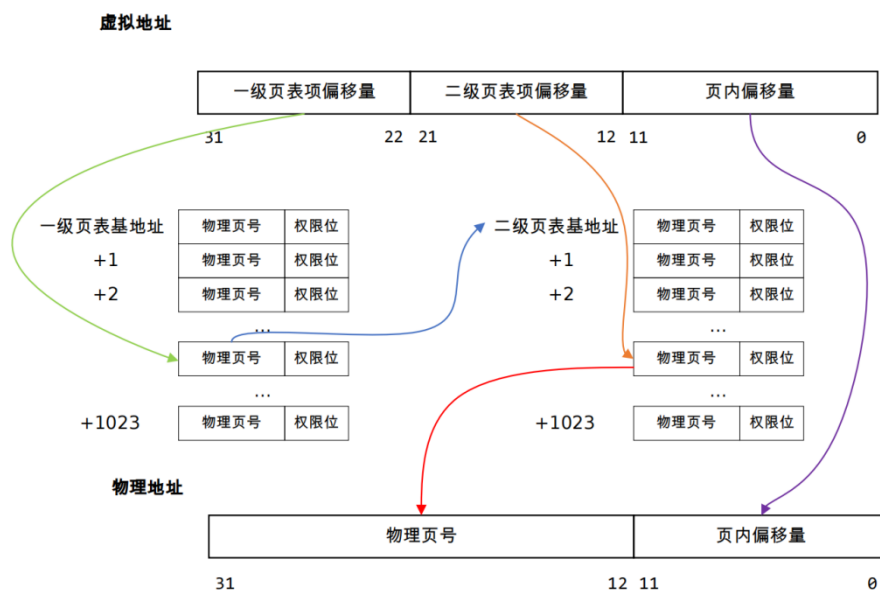


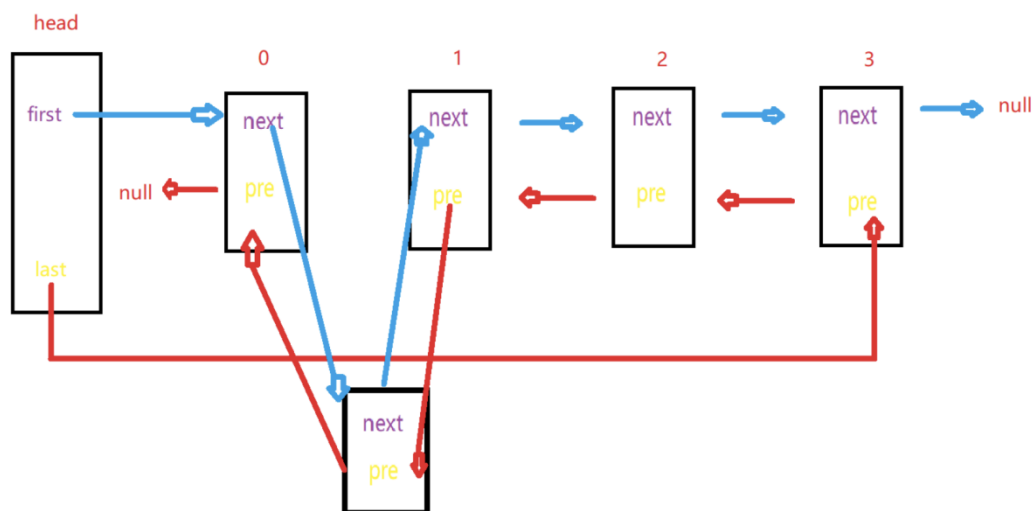
图 2.4: walk & map(insert) 系列函数图解流程



指导书的解

在 exercise 2.2 里 listlm.next.pre, 这个有点复杂,但总的来说其实质就是 le_prev = &le_next。
具体可以看这张图就可以知道 exercise2.2 需要做的

点击查看图片来源



https://blog.csdn.net/weixin_45728346

在 exercise2.8 看上去好像只需要补指令但重点是理解 `tlbp` 和 `tlbwi` 两条指令的含义以及大致的执行流程

体会与感谢

在这次的实验，是我目前做的全部实验耗的时间是最长的应该做了 3-4 天，相比前两个 lab，这更考的是我的基本功，在做这个 lab 的时候我一直都有一个错觉，就是我一直都以为我都懂了，然后写了，不对，又懂了，在这次 lab 我可以分为两个部分来吧，第一部份是完全在读懂代码，读了一遍又一遍感觉自己没太看懂这部分占了我很长时间，到第二部分开始写的，自己的基本功又不稳定，在指针和语法这方面还是需要加强，在这个 lab 我了解到内存的管理方法，不同的方法不同的表达，比如链表和两级页表的处理也不一样，还有怎么重填 TLB 这个也很重要，我觉得在后续的 lab 已经要学会读懂代码，怎么读懂源代码并应用起来，可能尝试自己在这个函数再继续扩展，前两次的 lab 都发挥得不是很好，更多的是自己心态和基础问题，希望在日后的 lab 可以更努力学会操作系统这门课吧。