Lab2

班级: 222111

学号: 22373340

姓名: 詹佳博

思考题

Thinking 2.1

• 虚拟地址;虚拟地址。

Thinking 2.2

请从可重用性的角度,阐述用宏来实现链表的好处。

- C语言中**没有泛型**这个概念。在学习 Java 的时候,我了解到**泛型**能够让我们更加面向需求的来运用相关的数据结构类型。结合大一上数据结构的经历,我经常会碰到要使用不同数据类型的链表、栈等,这个时候就得**定义多个结构体**,甚至操作函数代码也得**重复写多遍**。
- 用宏实现链表,可以巧妙的实现"泛型",也可以提高代码的效率。

请你查看实验环境中的 /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。

- 单向链表在环境中分为有尾和无尾。单向链表的插入操作的时间复杂度为O(1),删除操作的时间复杂度为O(n);
 - o SLIST 为单向无尾链表。进行插入操作时只需要将新插入节点的指针 field.sle_next 指向要插入的位置的下一个节点,并将要插入的上一个节点的指针 field.sle_next 指向它即可;删除时跟插入操作基本相反。
 - o STAILQ 为单向有尾链表。它的删除与插入操作相比 SLIST 多了对尾部的操作,在处理时多了if是尾部的判断,其余基本相同。
- CIRCLEQ 为循环链表。它的结构相比单向链表多一个最后一个节点的指针指向头节点。循环链表的插入操作的时间复杂度为O(1),删除操作的时间复杂度为O(n);
- 双向链表的插入操作的时间复杂度为O(1), 删除操作的时间复杂度为O(1);

Thinking 2.3

```
C:
struct Page_list{
    struct {
        struct Page *le_next;
        struct Page **le_prev;
        } pp_link;
        u_short pp_ref;
    }* lh_first;
}
```

Thinking 2.4

- ASID唯一标识每个进程,并用于为该进程提供地址空间保护,当TLB尝试解析VPN时,它确保当前正在运行的进程的ASID与VPN关联的ASID匹配。如果ASID不匹配,则将该尝试视为TLB未命中。
 ASID 使得操作系统能够在不同进程之间快速切换,同时保持各自的地址映射信息。
- ASID是一个8位的字段。这一意味着理论上有 2⁸ 即 256 个不同的地址空间,也即最大数量为 256。

Thinking 2.5

- tlb_invalidate 调用了 tlb_out
- tlb_invalidate 函数实现更新页表中虚拟地址对应的页表项的同时,将 TLB 中对应的旧表项无效化。

```
LEAF(t1b_out)
.set noreorder
   mfc0 t0, CP0_ENTRYHI # 从CP0取EntryHi数据,写入t0
         a0, CP0_ENTRYHI # 往CP0的EntryHi写入数据a0
   mtc0
                          # 待前面指令退出流水线
   nop
                          # 根据EntryHi中的Key查找对应的旧表项,将表项的索引存
   t1bp
入Index
                          # 待前面指令退出流水线
   nop
   mfc0 t1, CP0_INDEX # 从CP0取Index数据,写入t1
.set reorder
       t1, NO_SUCH_ENTRY # 如果Index数据小于0,则不存在条目,跳转到
   bltz
NO_SUCH_ENTRY
.set noreorder
   mtc0 zero, CP0_ENTRYHI
   mtc0 zero, CPO_ENTRYLOO # 存在条目的情况下,向EntryHi, EntryLoO和EntryLo1
写入0
   mtc0 zero, CP0_ENTRYLO1 #
                          # 待前面指令退出流水线
   nop
   tlbwi
                          # 将EntryHi、EntryLo0和EntryLo1中的0写入索引指定
的表项
.set reorder
NO_SUCH_ENTRY:
   mtc0 t0, CP0_ENTRYHI # 往CP0的EntryHi写入数据t0
                          #返回
   j
          ra
END(tlb_out)
```

Thinking A.1

• 三级页表的基地址为 PT_{base} ,则中间页表基地址为

 $(PT_{base} >> 12)*8 + PT_{base} = PT_{base} >> 9 + PT_{base}$,

```
PT_{base} >> 18 + PT_{base} >> 9 + PT_{base}

• 中间页表基地址为PT_{base} >> 9 + PT_{base}, 三级页表页目录基地址为 PT_{base} >> 18 + PT_{base} >> 9 + PT_{base}, 位移距离为PT_{base} >> 18, 故而有 (PT_{base} >> 18)/4K = PT_{base} >> 30项。 故而映射到页目录自身的页目录项(自映射)为 (PT_{base} >> 18 + PT_{base} >> 9 + PT_{base}) + PT_{base} >> 30 * 8, 也就是 PT_{base} >> 27 + PT_{base} >> 18 + PT_{base} >> 9 + PT_{base} >> 9 + PT_{base}
```

则**三级页表页目录基地址**为 $(PT_{base} >> 9 + PT_{base}) >> 9 + PT_{base}$,<mark>也就是</mark>

Thinking 2.6

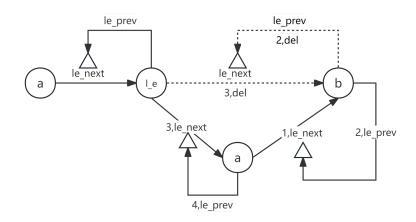
简单了解并叙述 X86 体系结构中的内存管理机制,比较 X86 和 MIPS 在内存管理上的区别。

- 在X86体系结构中,内存管理主要通过两种机制实现:分段和分页。分段机制允许将内存划分为不同的逻辑段,每个段由一个段址、段长和一组属性定义。分页机制则允许操作系统将物理内存划分为固定大小的页,并将这些页映射到地址空间,从而实现虚拟内存的概念。
- TLB的处理不同:在MIPS中,TLB不命中会触发TLB Miss异常,由内核处理;而在X86中,TLB不命中是由硬件MMU处理并填充,直接输出物理地址。

页表大小不同: MIPS为每个进程分配一个页给PDE,而X86的PDE大小依赖于进程代码、数据的大小。

难点分析

1. 在对 queue.h 进行阅读时,会觉得非常难读,并且实现相关的 macro 函数会感到比较困难。可能和之前从没有接触过宏函数编写有关。但是在模仿和伪代码的帮助下,以及必要的图解:



- 2. 指针太多,导致运用 [queue.h] 中的函数的时候有些模糊不清。但是采用面向对象的思维并且动手将宏代码的内容带入就解决了代码编写问题。在理解上抽象理解反而通俗易懂。
- 3. 在编写 [pgdir_walk] 函数时,错误理解题图,导致**索引的位移距离弄混**,之前有的左移了有的没左移4位,最后理清思路决定先全部划归为**指针引用**,后进行**下标索引**统一不左移。
- 4. 同时, pgdir_walk 函数要深刻理解"CPU发出的地址均为虚拟地址,因此获取相关物理地址后,需要转换为虚拟地址再访问"这句话。

实验体会

- 1. 这次lab用时比较长,尤其是Lab2_2部分的学习周期占主要地位。理解的过程十分曲折,在 pgdir_walk 函数部分停留许久,对 VA,PA,Page的转换有些懵圈。根据指导书的提示和分析才 艰难走完这段路。
- 2. 对内存管理机制有了初步的认识,虽然不能独立构建起完整的内存管理体系,但是能够知道他的框架,并对其中重要的部分比如**CPU仅操作虚拟地址,内核启动对内存的基本初始化,物理内存的数据结构,虚拟内存的二级页表机制,访存TLB重填**有一定掌握。
- 3. 在访存TLB重填部分,实验代码量很少,且比较容易实现,故而还是很顺利的完成了练习。但是其 汇编代码和c语言的流程还是相对比较复杂,目前还是能跟着指导书的分析一步步看懂和理解。
- 4. exam和extra和课上内容联系密切,extra写的有点冗杂但是好歹过了,有点为Lab1的extra没过而痛心疾首了。。。