LAB1实验报告

thinking2.1

> 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?

在编写的c程序中，指针变量存储的地址是虚拟地址。mips中的lw和sw也是虚拟地址。

thinking2.2

> 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。

由于c语言中没有c++语言的泛型语法，而在实际操作中可能需要能够快速适用于各种结构体中的链表结构，因而需要通过某种方式来实现这一点。用宏来实现链表本质上是一种自动化的代码生成手段，可以针对各种拥有统一结构和不同类型的链表节点和链表表头应用同一套处理方式。

从一个实例出发，当用户需要一个存储int类型数据的链表和一个存储长度为10的int数组的链表时，必然要定义两个存储了数据的结构体，在c语言中由于相关对齐机制，数据域所占的空间不同，无法通过指针类型转化来实现用同一个next、prev等方法同时支持两个链表的操作，因而需要针对这两种链表分别编写相关操作函数，大大增加了重复性的工作。

此时，如果使用宏来进行链表编写，则可以将操作转化为有参数宏，并在预处理阶段转化为操作对应链表的指令序列。

虽然如此，不可忽视的是以课程组提供的宏为例，他并没有充分利用函数机制，在拥有较快的速度的同时增大了指令量，当相关宏被频繁调用时会使得编译后的程序占用更多的空间，这是其局限性之一。

> • 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。

本实验的双向链表由于可以操作上一个链表项的next指针指向，因此可以快速实现删除操作，插入和删除的时间复杂度都是O(1)。

提到的头文件是linux提供的头文件，其中定义了双向链表LIST、单向链表SLIST、循环队列CIRCLEQ。

单向链表的项后插入时间复杂度为O(1)，删除的时间复杂度是O(n)，结构如下：

```
158 #define SLIST_ENTRY(type)                                               \
159 struct {                                                               \
160         struct type *sle_next;  /* next element */                      \
161 }
```

双向链表的实现与课程组提供的类似。

循环链表的插入和删除时间复杂度也是O(1)，结构如下：

```
479 #define CIRCLEQ_ENTRY(type)                                             \
480 struct {                                                                \
481         struct type *cqe_next;          /* next element */              \
482         struct type *cqe_prev;          /* previous element */          \
483 }
```

thinking2.3

正确结构是C，list指向第一个链表项，链表项中有数据域和链表项域。

thinking2.4

> • 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指的是Address Space Identifier，既地址空间标识符。

在现代的多任务操作系统中，通常使用相同的程序地址空间，并映射到不同的物理地址，这被称为使用不同的“地址空间”。不同的进程使用自己独立的的地址空间，而操作系统需要通过TLB使不同地址对相同地址的访问相互隔离。具体而言，当进程通过TLB访问内存时VPN相同十分常见，而如果当前进程的EntryHi中的ASID与TLB中的不同，则视为不命中，进而触发TLB缺失进而访问当前进程的页表进行进一步处理。如果没有ASID，则需要在进程切换时刷新所有的TLB信息，这是较为低效的且可能会在频繁切换时降低TLB的工作效率，否则则会面临任务切换后地址空间映射错误的问题。

如果操作系统不进行特别的ASID再分配优化，则逻辑层面上来说6位的ASID段只能提供64个不同的地址空间。然而操作系统的实现层面上操作系统可以支持更多的地址空间，只需要在64个ASID段全部分配之后依次清除所有进程的ASID记录并清空TLB，进而为新到来的任务重新分配ASID。如果操作系统实现了如上操作，便可以在内存容量允许的范围内支持更多的地址空间，只是会降低切换和访存的速度。

thinking2.5

tlb_invalidate调用tlb_out。

触发针对特定asid和va的tlb表项删除。

解释如下：

```
 3 LEAF(tlb_out)
 4 .set noreorder
 5         mfc0    t0, CP0_ENTRYHI //存储ENTRYHI内容到t0
 6         mtc0    a0, CP0_ENTRYHI //将要清除的表项对应的KEY写入ENTRYHI
 7         nop
 8         /* Step 1: Use 'tlbp' to probe TLB entry */
 9         /* Exercise 2.8: Your code here. (1/2) */
10         tlbp                    //开始查找目标KEY对应的tlb项是否存在
11         nop
12         /* Step 2: Fetch the probe result from CP0.Index */
13         mfc0    t1, CP0_INDEX   //查找结果的索引存储到t1，没查找到则最高位置1
```

```
.set reorder
        bltz    t1, NO_SUCH_ENTRY   //如果没查找到，既t1中存储的INDEX结果的最高位是1，跳转到NO_SUCH_ENTRY
.set noreorder
        mtc0    zero, CP0_ENTRYHI   //如果查到了，将ENTRYHI和ENTRYLO置零
        mtc0    zero, CP0_ENTRYLO0  //同上
        nop
        /* Step 3: Use 'tlbwi' to write CP0.EntryHi/Lo into TLB at CP0.Index  */
        /* Exercise 2.8: Your code here. (2/2) */
        tlbwi                       //写入清零INDEX中查找到的索引值位置对应的TLB项目
.set reorder

NO_SUCH_ENTRY:                      //如果没查找到则跳过上述步骤，直接到这里
        mtc0    t0, CP0_ENTRYHI     //恢复当前进程的ENTRYHI，其中保存着对应的ASID
        j       ra                  //跳回调用点
END(tlb_out)
```

thinking2.6

mips的内存管理较为容易，二级页表的结构也比较简单。

而x86的内存管理机制较为复杂。较为重要的是其使用段页式地址映射机制，首先将逻辑地址通过分段机制映射到线性地址，再通过页映射机制将线性地址映射到物理地址。

在分段机制中，不同的指令性质对应不同的段寄存器，根据段寄存器的内容找到SegmentDescriptor找到基地址，将基地址与合法的指令发出的地址，判断越界和越权后得到线性地址。

在页映射机制中，首先从CR3寄存器获取页目录，以线性地址的Directory段为下标获取页面表，以线性地址的Table段为下标在页面表中获取页面描述项，最后将页面描述项中的页面基地址与线性地址中的偏移位段组合得到物理地址。

同时x86的内存管理机制中包含了虚拟页面的相关操作，MMU负责地址转换并触发缺页中断，通过相关中断处理程序处理缺失页面。

thinkingA.1

映射到PTbase的中间页目录项在第(PTbase>>12)个页表项处，每个页表占8字节，既映射到PTbase的页表项位置在PTbase+PTbase>>9处，从这里开始的4k大小页面是二级目录，映射着512个三级目录。页目录基地址处的页目录项应当映射映射到映射三级页表的中间目录的起始地址，故位于PTbase+(PTbase+PTbase>>9)>>9=PTbase+PTbase>>9+PTbase>>18。

映射到页目录自身的页目录项只需要重复上述步骤，该页目录项位于PTbase+PTbase>>9+PTbase>>18+PTbase>>27

难点分析

本次作业中，难点在于理解各个函数和结构体在整体系统内核结构中的作用。

1. 其中对于queue.h中LIST结构的理解是关键之一。

listitem是一个指向链表节点的指针，节点包含任何自定义的数据段和一个field作为自定义参数链表项段。field中包含的两个指针一个是指向下一个链表节点的指针next，另一个是指向上一个节点中的next的指针。因此，这个双向链表结构实际上不能在O(1)的时间内得到链表的上一项。这其实是我不太理解原因的部分，也许是因为这样可以加快常用操作的耗时吧。

2. 本次作业主要在kern/pmap.c中工作，其中各个函数的作用的理解比较重要。
    - 22 void mips_detect_memory()

      通过我看不懂的方式得到了memsize，代表总内存字节容量，很小只有64MB。

      通过地板除BY2PG得到可以分配的页面最大值npage

      输出内存kB大小和对应的页面总数
    - 38 void *alloc(u_int n, u_int align, int clear) {
        - PADDR(x)可以获取虚拟地址对应的物理地址。

      freemem对应着可分配内存的初始位置的虚拟内存地址。

      首先拿到链接脚本中定义的end位置，如果第一次调用则初始化freemem到end位置，这是为了避开系统的代码和数据位置，从0x80400000开始分配。

      free对齐到align并记录初始位置到alloced_mem，然后free-加上要分配的内存量，检查对应的物理地址是否超过内存上限。

      根据clear初始化对应位置，并返回记录的初始位置
    - 73 void mips_vm_init() {

      根据内存大小对应的页表项数量，初始化对应的struct Page数组以建立页面管理机制。
    - 89 void page_init(void) {

      初始化page_free_list，由于页表项已经占用了一部分内存，故将占用的内存部分对应的页面管理结构体的pp_ref位置设置成1，其他设置成0。0对应的空闲页面会被插入page_free_list，这个list表头保存在数据段，既物理地址0x400000前。
        - page2pa(struct Page *)通过全局变量pages获取指针对应的页面索引，左移12位得到对应的物理地址

      由于当前代码运行在seg0，对应0x0开始的位置，40_0000前对应的是代码和数据，故有16384-1024个页面是能够自由分配的，又因为pages占用了一部分(48)个页面，故只有15312个页面可以自由分配。
    - 131 int page_alloc(struct Page **new) {

      传入存储返回的内存控制块的**new

      分配一个空闲的物理内存页面并初始化。

      如果pfl为空，则报错返回-E_NO_MEM

      否则提取到pfl第一项，清零，new对应指针保存内存控制块的位置。
    - 157 void page_free(struct Page *pp) {

      传入内存控制块，判断是不是已经在pagefreelist中了

      插回到pfl中
    - 181 static int pgdir_walk(Pde *pgdir, u_long va, int create, Pte **ppte) {

关键函数，传入的pgdir是一个二级列表的页目录指针，va是目标虚拟地址，create决定找不到时是否创建，**ppte保存返回的页表项指针。

其中Pde和Pte都是unsigned long类型，用来保存页表项和页目录项。

- PDX 取31到22位，对应页目录项索引。 Directory
- PTX 取21到12位，对应页表项索引。 Table

首先获取pgdir中对应va的页目录项，判断这个页目录项是否有效。

- PTE_V 页项中的validate 位

如果不有效，判断是否要创建，创建则分配一个新的页面，用来存储映射4MB内存的页表，页表大小为1024*4 = 4KB，恰好为一页，所以用page_alloc(&pp)分配。分配到的pp页控制块在转化为物理地址后设置D、V为为1并保存回页目录项中。

现在，页目录项已经保证有效，既指向了一个有效的页表。

- PTE_ADDR 通过置零低12位获取页表项对应的地址。
- KADDR 获取物理地址对应的虚拟地址。

PTEADDR(pgdir_entryp)得到页表的物理地址，由于内核代码所有访存操作需要通过0x8开头的虚拟地址，故使用`( (Pte*) KADDR(PTE_ADDR(*pgdir_entryp)) )`获取指向该页表的指针，并加上页表项索引从而使ppte指向页表项。

- 232 int page_insert(Pde *pgdir, u_int asid, struct Page *pp, u_long va, u_int perm) {

  找到pte，重填tlb，设置perm和PTE_V，并pp对应的pp_ref加1.

  如果找到有效页表项，则设置perm后推出，否则创建正确的页表项。

- 270 struct Page *page_lookup(Pde *pgdir, u_long va, Pte **ppte) {
  - pa2page 获取pa对应的物理页面控制块。

  尝试查找va对应的页面控制块，由于pgdir_walk获取到的是页表项指针，需要使用pa2page转化。返回控制块并保存页表项指针到ppte

实验体会

在本次实验中我们构建了两级页表管理机制。过程可以归纳为：

1. 将内存按照页面大小分配页面控制块。
2. 设置自映射的页控制块pp_ref为1。
3. 编写查找va对应的页表项、叶控制块的函数。

在这个过程中，我深入了解了相关代码的实现原理，并对内存如何分配有了一部分认识。

同时，对于如何利用宏函数实现链表等数据结构也有了初步的认识，对于宏的嵌套调用解析也进行了相关的学习。

特别是对于系统内核设计中“指针的指针”相关使用有了较为深刻的认识。