BUAA-OS-lab2

发表于 2023-04-10 | 更新于 2023-06-19 | 养德（学习）
| 字数总计: 7.2k | 阅读时长: 27分钟 | 阅读量: 1570

Lab2实验报告

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？

- 解：
  - 在编写的 C 程序中，指针变量中存储的地址是**虚拟地址**；
  - MIPS 汇编程序中 lw 和 sw 使用的是**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，分析三者在插入与删除操作上的性能差异。

- 解：
  - 1. 各类结构体都可以使用 `queue.h` 的宏简化代码量：
    - 一开始写**Exercise2.2**的时候没有明白 `field` 字段设置的意义，觉得传入了 `elm` 参数以后， `field` 应该是作为 `elm` 的成员变量名，不应该作为可变参数传入，显得有点多此一举，忘记了宏定义除了可以替换成实参，还可以做字符串字面量的替换。
    - 写到**Exercise2.3**才意识到 `field` 不是变量参数，而是 `pp_link` 这个字符串。
    - 后来思考之所以在 `queue.h` 的宏里面这样定义，应该是因为这里的宏并不止服务于 `Page` ，所以从工程角度来看， `field` 仍然属于一种变量， `field` 的设置提高了宏的可重用性，因为可以兼顾所有结构体的所有成员变量名。
  2. 复杂宏(如 `LIST_INSERT_TAIL` 、 `LIST_INSERT_BEFORE`)大量使用简单宏(如 `LIST_FIRST` 、 `LIST_NEXT`)，可重用性强
  3. 可读性强，简化代码量，易于维护
  - 单向链表

    对于单纯的插入和删除操作只有**O(1)的时间复杂度，但是对于任意第 `i` 个元素的插入和删除操作来说，则需要从头遍历一遍故而时间复杂度会上升到O(n)**;
    - 双向链表

      对于任意第 `i` 个元素的插入和删除操作时间复杂度都只有**O(1)**;
    - 循环链表
      - 单向循环链表

        和单向链表一样，对于任意第 `i` 个元素的插入和删除操作来说，时间复杂度会为**O(n)**;
      - 双向循环链表

        和双向链表一样，对于任意第 `i` 个元素的插入和删除操作来说，时间复杂度会为**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。

- 解：
  - 正确展开应该是 C：

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

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。

- 解：
  - 由于在多线程系统中，对于不同进程，相同的虚拟地址各自占用不同的物理地址空间，所以同一虚拟地址通常映射到不同的物理地址。因此TLB中装着不同进程的页表项，ASID用于区别不同进程的页表项。

    没有ASID机制的情况下每次进程切换需要地址空间切换的时候都需要清空TLB。
  - ASID6位，容纳64个不同进程。

Thinking 2.5

- 答案见**Exercise 2.8**

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。

- 解：
  - X86 体系结构中的内存管理机制
    - 通过**分段**将**逻辑地址**转换为**线性地址**，通过**分页**将**线性地址**转换为**物理地址**。
    - 逻辑地址由两部分构成，一部分是**段选择器**，一部分是**偏移**。
    - **段选择符**存放在段寄存器中，如CS（存放代码段选择符）、SS（存放堆栈段选择符）、DS（存放数据段选择符）和ES、FS、GS（一般也用来存放数据段选择符）等；
    - 偏移与对应段描述符中的基地址相加就是**线性地址**。
    - 操作系统创建全局描述符表和提供逻辑地址，之后的分段操作x86的CPU会自动完成，并找到对应的线性地址。
    - **从线性地址到物理地址的转换是CPU自动完成**的，转化时使用的Page Directory和Page Table等需要操作系统提供。
  - X86 和 MIPS 在内存管理上的区别
    - TLB不命中：
      - MIPS触发TLB缺失和充填，然后CPU重新访问TLB
      - x86硬件MMU索引获得页框号，直接输出物理地址，MMU充填TLB加快下次访问速度
    - 分页方式不同：
      - 一种MIPS系统内部只有一种分页方式
      - x86的CPU支持三种分页模式
    - 逻辑地址不同：
      - MIPS地址空间32位
      - x86支持64位逻辑地址，同时提供转换为32位定址选项
    - 段页式的不同：
      - MIPS同时包含了**段**和**段页**式两种地址使用方式
      - 在x86架构的保护模式下的内存管理中，**分段是强制**的，并不能关闭，而**分页是可选**的；

Thinking A.1

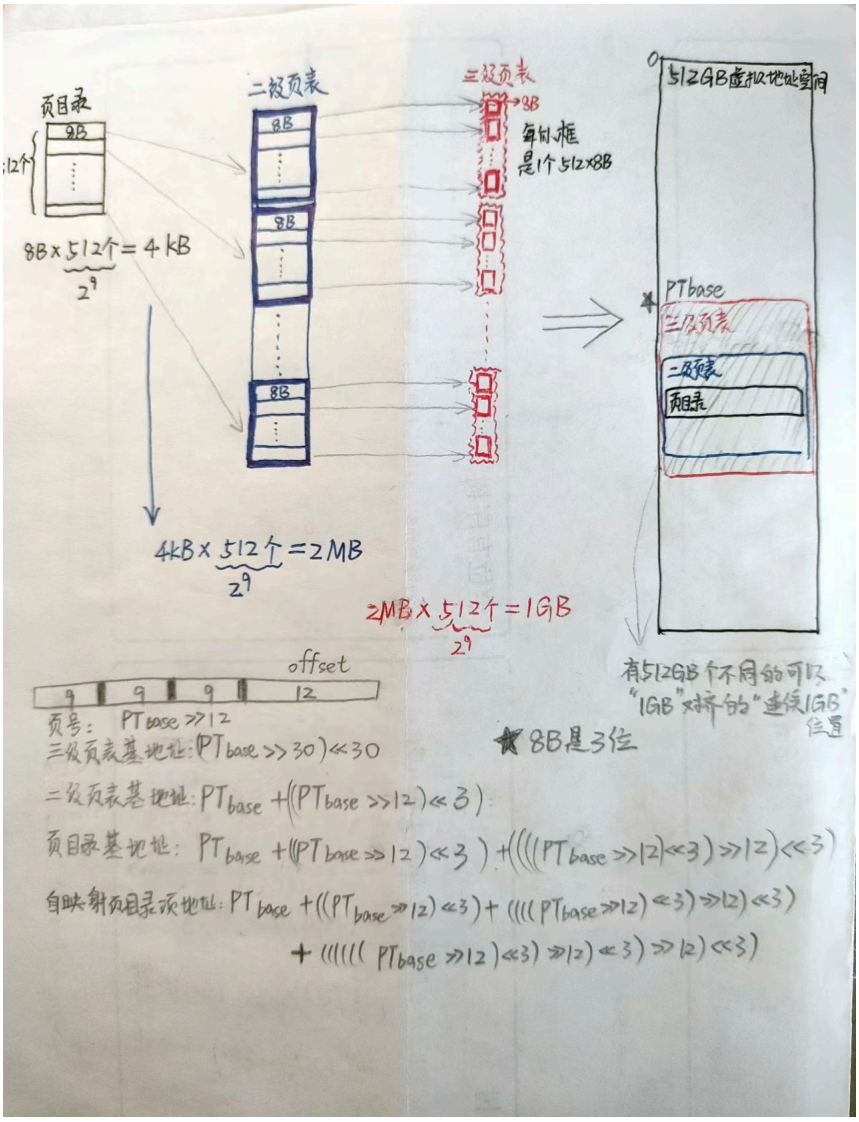
在现代的 64 位系统中，提供了 64 位的字长，但实际上不是 64 位页式存储系统。假设在 64 位系统中采用三级页表机制，页面大小 4KB。由于 64 位系统中字长为 8B，且页目录也占用一页，因此页目录中有 512 个页目录项，因此每级页表都需要 9 位。 因此在 64 位系统下，总共需要 3 × 9 + 12 = 39 位就可以实现三级页表机制，并不需要 64 位。

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。

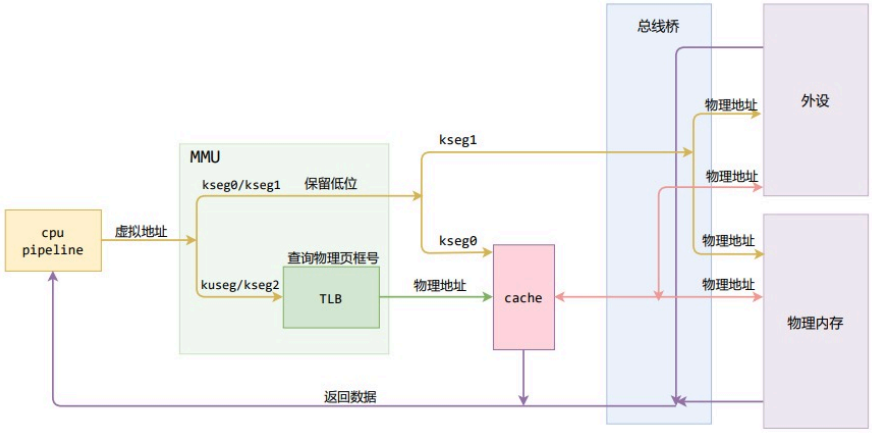
• 映射到页目录自身的页目录项（自映射）。

- 解：
  - 这里贴一张周伯阳大佬的博客，佬佬解释得非常清晰：AugetyVolta's Blog 页目录自映射

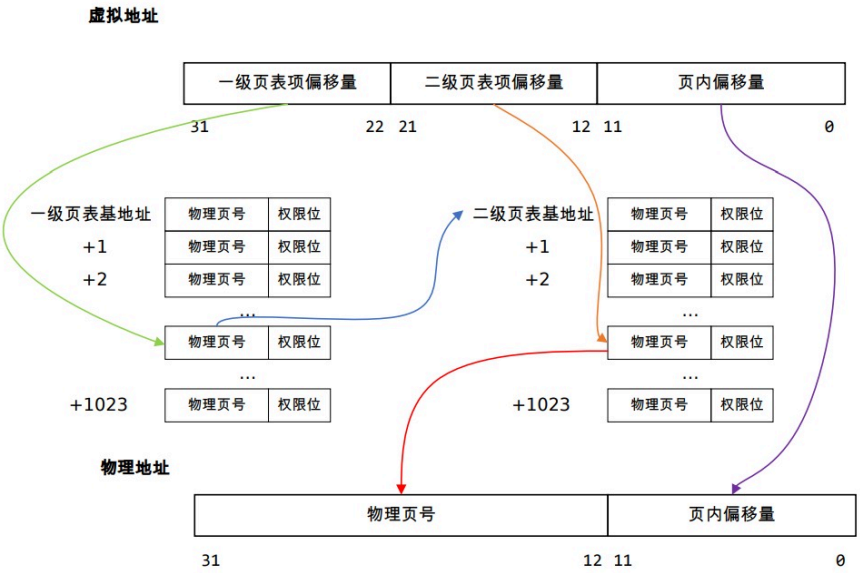


难点分析

- cpu-tlb-memory 关系：



- 两级页表结构：



实验体会

Exercise 2.1

> 请参考代码注释，补全 mips_detect_memory 函数。 在实验中，从外设中获取了硬件可用内存大小 memsize，请你用内存大小 memsize 完 成总物理页数 npage 的初始化。

- 由MIPSR3000文档得一个页表大小是4k，则 `npage = memsize / 4096;` 但是，由于定义有 `PGSHIFT` ，则直接 `npage = memsize >> PGSHIFT;`

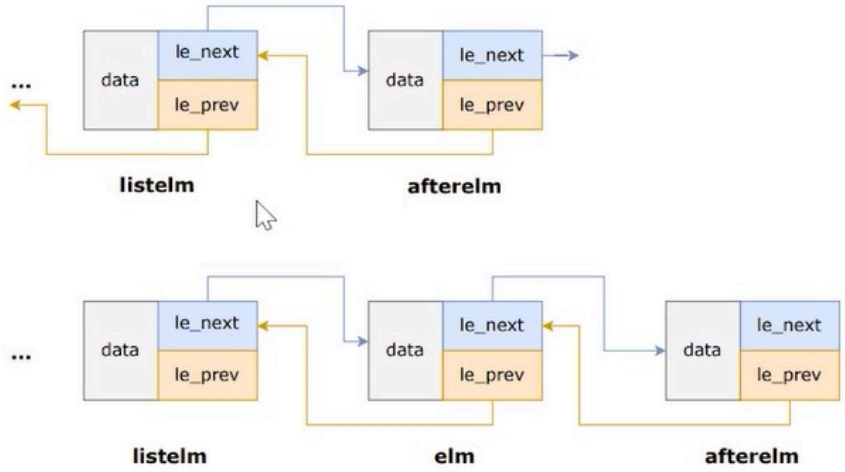
Exercise 2.2

> 完成 include/queue.h 中空缺的函数 LIST_INSERT_AFTER。 其功能是将一个元素插入到已有元素之后，可以仿照 LIST_INSERT_BEFORE 函数来实现。

- 理清结构体中的关系：

  指向下一个元素的指针 le_next ；

  指向前一个元素链表项 le_next 的指针 le_prev 。【le_prev 是一个指针的指针!】



> 注意：每一行都要加上"\" 哟~

```
/*
 * Insert the element 'elm' *after* 'listelm' which
 * name is the link element as above.
 *
 * Hint:
 * Step 1: assign 'elm.next' from 'listelm.next'.
 * Step 2: if 'listelm.next' is not NULL, then assig
 * Step 3: assign 'listelm.next' from a proper value
 * Step 4: assign 'elm.pre' from a proper value.
 */
#define LIST_INSERT_AFTER(listelm, elm, field)
        /* Exercise 2.2: Your code here. */  \
        do {
                LIST_NEXT((elm), field) = LIST_NEXT(
                if (LIST_NEXT((listelm), field) != N
                        LIST_NEXT((listelm), field)-
                }
                LIST_NEXT((listelm), field) = (elm);
                (elm)->field.le_prev = &(LIST_NEXT((
        } while (0)
    /*do {
                elm的le_next = elm在链表中的下一个元素
                if ( elm不是链表尾) {
                        elm之后元素的le_prev = elm的l
                }
                listelm的le_next = elm的地址;
```

```
27                  elm的le_prev = listelm的le_next的地址
28          }while (0);*/
```

Exercise 2.3

> 完成 page_init 函数。 请按照函数中的注释提示，完成上述三个功能。此外，这里也给出一些提示：
>
> 1. 使用链表初始化宏 LIST_INIT。
> 2. 将 freemem 按照 BY2PG 进行对齐（使用 ROUND 宏为 freemem 赋值）。
> 3. 将 freemem 以下页面对应的页控制块中的 pp_ref 标为 1。
> 4. 将其它页面对应的页控制块中的 pp_ref 标为 0 并使用 LIST_INSERT_HEAD 将 其插入空闲链表。

- 我的答案：

```
1   void page_init(void) {
2           /* Step 1: Initialize page_free_list. */
3           /* Hint: Use macro `LIST_INIT` defined in in
4           /* Exercise 2.3: Your code here. (1/4) */
5           LIST_INIT((&page_free_list));
6           /* 初始化空闲页表，及申请一块没有用过的地址*/
7
8           /* 以下都是为了初始化 pages[] 页面控制块数组 */
9           /* Step 2: Align `freemem` up to multiple of
10          /* Exercise 2.3: Your code here. (2/4) */
11          freemem = ROUND(freemem, BY2PG);
12          /* freemem = ROUND(待对齐地址freemem ，对齐方i
13          /*将当前已分配的地址上界 freemem 使用 ROUND 宏i
14          /* ROUND 在 types.h 中，作用是对第一个参数传入t
15
16          /* Step 3: Mark all memory below `freemem` a
17          /* Exercise 2.3: Your code here. (3/4) */
18          int size = PADDR(freemem) / BY2PG;
19          /* size表示已经分配出去的物理页面的数量
20             size = 已分配的物理内存 / 页面大小
21             由于 freemem 为内核虚拟地址，则需要使用 PADD
22             PADDR在mmu.h中，作用是将传入参数减去ULIM，得
23             从0x80010000开始是第一个物理页面的开始地址，
24             回看kernel.lds，可见end被赋值为0x80400000,s
25          int i;
26      for(i = 0; i < size; i++) {
27          pages[i].pp_ref = 1;
28                  /*将已经分配出去的物理页控制块 pages[i]
```

```
    }
        /* Step 4: Mark the other memory as free. */
        /* Exercise 2.3: Your code here. (4/4) */
        for(; i < npage; i++) {
        pages[i].pp_ref = 0;
                /*将未分配出去的物理页控制块 pages[i] 
        LIST_INSERT_HEAD((&page_free_list), (pages +
                /*LIST_INSERT_HEAD((空闲链表的地址), 
                /*使用 LIST_INSERT_HEAD宏将空闲页面的
    }
}
```

Exercise 2.4

> 完成 page_alloc 函数。 在 page_init 函数运行完毕后，在 MOS 中如果想申请存储空间，都是通过这个函 数来申请分配。该函数的逻辑简单来说，可以表述为：
>
> 1. 如果空闲链表没有可用页了，返回异常返回值。
> 2. 如果空闲链表有可用的页，取出第一页；初始化后，将该页对应的页控制块的 地址放到调用者指定的地方。
>
> 填空时，你可能需要使用链表宏 LIST_EMPTY 或函数 page2kva。

- 我的答案：

```
/* 将 page_free_list 空闲链表头部页控制块对应的物理页面
 * 将其从空闲链表中移除，并清空对应的物理页面，
 * 最后将 pp 指向的空间赋值为这个页控制块的地址。
 */
int page_alloc(struct Page **new) {
        /* Step 1: Get a page from free memory. If f
        struct Page *pp;
        /* Exercise 2.4: Your code here. (1/2) */
        if(LIST_EMPTY((&page_free_list))) {
                return -E_NO_MEM;
        } else {
                pp = LIST_FIRST((&page_free_list));
                /*用LIST_FIRST将空闲链表头部页控制块对
        }
        LIST_REMOVE(pp, pp_link);
        /*将其从空闲链表中移除*/

        /* Step 2: Initialize this page with zero.
         * Hint: use `memset`. */
        /* Exercise 2.4: Your code here. (2/2) */
```

```
21          memset((void *)page2kva(pp), 0, BY2PG);
22          /*清空对应的物理页面 memset((void *)新申请的空
23            page2kva从页面控制块，获得其对应的内核虚拟地址
24          *new = pp;
25          return 0;
26  }
```

Exercise 2.5

> 完成 page_free 函数。 提示：使用链表宏 LIST_INSERT_HEAD，将页结构体插入空闲页结构体链表。

- `void assert(int expression);` 允许诊断信息被写入到标准错误文件中。换句话说，它可用于在 C 程序中添加诊断。
  - `expression` – 这可以是一个变量或任何 C 表达式。如果 `expression` 为 TRUE，assert() 不执行任何动作。如果 `expression` 为 FALSE，assert() 会在标准错误 stderr 上显示错误消息，并中止程序执行。

```
1   void page_free(struct Page *pp) {
2           assert(pp->pp_ref == 0);
3           /*调用该函数的前提条件为 pp 指向页控制块对应的物
4            （即该物理页面为空闲页面）*/
5
6           /* Just insert it into 'page_free_list'. */
7           /* Exercise 2.5: Your code here. */
8           LIST_INSERT_HEAD((&page_free_list), (pp), pp
9           /*将其对应的页控制块重新插入到 page_free_list*/
10  }
```

Exercise 2.6

> 完成 pgdir_walk 函数。
>
> 该函数的作用是：给定一个虚拟地址，在给定的页目录中查找这个虚拟地址对应的物 理地址，如果存在这一虚拟地址对应的页表项，则返回这一页表项的地址；如果不存在这 一虚拟地址对应的页表项（不存在这一虚拟地址对应的二级页表、即这一虚拟地址对应的页目录项为空或无效），则根据传入的参数进行创建二级页表，或返回空指针。 注意，这里可能会在页目录表项无效且 create 为真时，使用 page_alloc 创建一个页 表，此时应维护申请得到的物理页的 pp_ref 字段。

- 我的答案：

```
/*将一级页表基地址 pgdir 对应的两级页表结构中 va 虚拟地址
 */
static int pgdir_walk(Pde *pgdir, u_long va, int cre
        Pde *pgdir_entryp;
        struct Page *pp;

        /* Step 1: Get the corresponding page direct
        /* Exercise 2.6: Your code here. (1/3) */
        pgdir_entryp = pgdir + PDX(va);
        /* 计算页目录项的指针
         * pgdir_entryp = 页目录基地址 + va所对的页目录
         * 等价于&pgdir[PDX(va)]*/

        /* Step 2: If the corresponding page table i
         * is set, create one. Set the permission bi
         * page directory.
         * If failed to allocate a new page (out of
        /* Exercise 2.6: Your code here. (2/3) */
        if(((*pgdir_entryp) & PTE_V) == 0) { //是否有
        if(create == 1) {
            if (page_alloc(&pp) < 0) { //表示没有多余
                            *ppte = NULL;
                                return -E_NO_MEM;
                        }
                        pp->pp_ref++;//维护申请得到的
            (*pgdir_entryp) = (PTE_D | PTE_V | page2
        }else{
            *ppte = NULL;
                        return -E_NO_MEM;
        }
    }
        /* (*pgdir_entryp)表示va对应的页目录项
       通过权限位 PTE_V 判断页目录的有效性;若页目录项无效
          if(((*pgdir_entryp) & PTE_V) == 0) {

              当create为1，表明此时的 pgdir_walk 用于
              所以当二级页表不存在的时候，需要创建二级
        if(create == 1) {
            if (page_alloc(&pp) < 0) { //表示没有多余
                            *ppte = NULL;
                                return -E_NO_MEM;
                        }
                        pp->pp_ref++;//维护申请得到的
            (*pgdir_entryp) = (PTE_D | PTE_V | page2
                        页目录项 = PTE_D | PTE_V | 二
                        根据分配的二级页表，填写页目录项
                        使用 page2pa(pp) 即可由页面控制
        }else{
```

```
49              *ppte = NULL;
50                      return -E_NO_MEM;
51          }
52      }
53           */
54
55          /* Step 3: Assign the kernel virtual address
56          /* Exercise 2.6: Your code here. (3/3) */
57          *ppte = (Pte *)KADDR(PTE_ADDR(*pgdir_entryp)
58          /* 将 va 虚拟地址所在的二级页表项的指针存储在 pp
59             *ppte = va 虚拟地址所在的二级页表项的指针
60                    = 二级页表基地址（指向二级页表的指针）
61
62                  二级页表基地址（指向二级页表的指针）：
63                          页目录项      ->    二级页表的物
64                          (*pgdir_entryp) 到 PTE_ADDR(
65
66                  va所对的二级页表项在二级页表的偏移：  PT
67          return 0;
68  }
```

Exercise 2.7

> 完成 page_insert 函数（补全 TODO 部分）。

- 我的答案：

```
1   /* 将一级页表基地址 pgdir 对应的两级页表结构中虚拟地址 v
2    */
3   int page_insert(Pde *pgdir, u_int asid, struct Page
4           Pte *pte;
5
6           /* Step 1: Get corresponding page table entr
7           pgdir_walk(pgdir, va, 0, &pte);
8           /*检查va是否存在映射，因为这里仅仅是检查，而不是
9
10          if (pte && (*pte & PTE_V)) { /*如果存在的映射
11                  /*判断现有映射的页 pa2page(*pte)和需要
12                    pa2page(*pte) 由于二级页表项的高20位
13                  if (pa2page(*pte) != pp) {
14                          /*如果不一样，则移除现有映射，
15                          page_remove(pgdir, asid, va)
16                  } else {
17                          /*如果一样，则只需要更新一下映
18                          tlb_invalidate(asid, va);
19                          /*但是要将 TLB 中缓存的页表项
20                          *pte = page2pa(pp) | perm |
```

```
                        return 0;
                }
        }

    /*页表项无效时，需要建立地址va到物理页面pp的映射，即

        /* Step 2: Flush TLB with 'tlb_invalidate'.
        /* Exercise 2.7: Your code here. (1/3) */
        tlb_invalidate(asid, va);
        /* 清除TLB中缓存的页表项*/

        /* Step 3: Re-get or create the page table e
        /* If failed to create, return the error. */
        /* Exercise 2.7: Your code here. (2/3) */
        /* 使用pgdir_walk 获取va对应的二级页表项的指针,
        if (pgdir_walk(pgdir, va, 1, &pte) == -E_NO_
        {
                return -E_NO_MEM;
        }

        /* Step 4: Insert the page to the page table
         * 'pp_ref'. */
        /* Exercise 2.7: Your code here. (3/3) */
        /*将虚拟地址va映射到页控制块pp对应的物理页面*/
        (*pte) = page2pa(pp) | perm | PTE_V;
        /*填写va所对的页表项：(*pte) = 页面控制块pp对应
                                页面控制块pp对应的物理页面
        (pp->pp_ref)++; /*维护物理页pp的pp_ref字段，由
        return 0;
}
```

Exercise 2.8

> 完成 kern/tlb_asm.S 中的 tlb_out 函数。该函数根据传入的参数（TLB 的 Key）找到对应的 TLB 表项，并将其清空。
>
> 具体来说，需要在两个位置插入两条指令，其中一个位置为 tlbp，另一个位置为 tlbwi。
>
> 因流水线设计架构原因，tlbp 指令的前后都应各插入一个 nop 以解决数据冒险。

- 我的答案：

```
/*tlb_asm.S*/
#include <asm/asm.h>

```

```
LEAF(tlb_out)
.set noreorder
        mfc0    t0, CP0_ENTRYHI
        /*把当前VPN和ASID存储到$t0，用于函数结束时恢复C
        mtc0    a0, CP0_ENTRYHI
        /*把调用函数时传入的VPN和ASID写进CP0_ENTRYHI*/
        nop
        /* Step 1: Use 'tlbp' to probe TLB entry */
        /* Exercise 2.8: Your code here. (1/2) */
        tlbp
        /*根据当前的CP0_ENTRYHI，查找TLB与之对应的表项，
        nop
        /* Step 2: Fetch the probe result from CP0.I
        mfc0    t1, CP0_INDEX
        /*把tlbp的执行结果记录到$t1中，供下面判断*/
.set reorder
        bltz    t1, NO_SUCH_ENTRY
        /*表示没有查到该表项*/
.set noreorder
        mtc0    zero, CP0_ENTRYHI   /*便于下面用tlbwi
        mtc0    zero, CP0_ENTRYLO0  /*便于下面用tlbwi
        nop
        /* Step 3: Use 'tlbwi' to write CP0.EntryHi/
        /* Exercise 2.8: Your code here. (2/2) */
        tlbwi
        /*把为零的CP0_ENTRYHI和CP0_ENTRYLO赋值回表项*/
.set reorder

NO_SUCH_ENTRY:
        mtc0    t0, CP0_ENTRYHI
        /*把原来的VPN和ASID赋值回CP0_ENTRYHI*/
        j       ra      /*跳回去*/
END(tlb_out)
```

Exercise 2.9

> 完成 kern/tlbex.c 中的 _do_tlb_refill 函数。

- 我的答案：

```
/* 该函数是TLB重填过程的核心，是根据虚拟地址va和ASID查找
 */
Pte _do_tlb_refill(u_long va, u_int asid) {
        Pte *pte;
        /* Hints:
         *  Invoke 'page_lookup' repeatedly in a loo
         *  with the virtual address 'va' in the cur
```

```
 *
 *  **While** 'page_lookup' returns 'NULL',
 *  allocate a new page using 'passive_alloc
 */

/* Exercise 2.9: Your code here. */
/* 在循环中调用 page_lookup 查找虚拟地址va在当前
while (page_lookup(cur_pgdir, va, &pte) == N
{
        /* 如果page_lookup返回NULL，表明pte找不
           用passive_alloc为va所在的虚拟页面分
        passive_alloc(va, cur_pgdir, asid);
        /* cur_pgdir是在pmap.c中定义的全局变量
}

return *pte;
}
```

Exercise 2.10

> 完成 kern/tlb_asm.S 中的 do_tlb_refill 函数。

- 见指导书P78

课上测试

知识点

- **遍历虚拟地址空间**

```
//已知Pde *pgdir
for (u_long i = 0; i < 1024; i++) { //遍历页目录的1024
        Pde *pde = pgdir + i; //第i个页目录项对应的虚拟
        if ((*pde) & PTE_V) { //第i个页表有效
                for (u_long j = 0; j < 1024; j++) {
                        Pte *pte = (Pte*)KADDR(PTE_A
                        if ((*pte) & PTE_V)) { //第j
                                /* some code*/
                /*例如：
                1.将(*pte)页表项映射到“某物理地址”，且
                (*pte) = ((某物理地址 / BY2PG) << 12)
                                        //上式作用等价
                                2.定向设置(*pte)页表项
                                (*pte) = ((*pte) | P
                                3.改变TLB中的(*pte)页
                                tlb_invalidate(asid,
                                    //tlb_invalidate
```

```
                */
                        }
                }
        }
}
```

- **pde (*pde) pte (*pte) PTE_ADDR PDX PTX**

```
//已知Pde *pgdir
Pde *pde = pgdir + i;
   /* pde:                   第i个页目录项对应的虚拟地
      (*pde):                 第i个页目录项内容
      PTE_ADDR(*pde):         第i个页目录项内容中的物理
      KADDR(PTE_ADDR(*pde)): 第i个页表的物理基地址*/
Pte *pte = (Pte*)KADDR(PTE_ADDR(*pde)) + j;
   /* pte:            第j个页表项对应的虚拟地址
      (*pde):         第j个页表项内容 */
u_long va =  (i << 22) | (j << 12) | offset ; // va
u_long i  = PDX(va);
u_long j  = PTX(va);
```

lab2-exam

对于给定的页目录 `pgdir`，统计其包含的所有二级页表中满足以下条件的页表项：

1. 页表项有效；
2. 页表项映射的物理地址为给定的 Page *pp 对应的物理地址；
3. 页表项的权限包含给定的权限 perm_mask。

答案

```
u_int page_perm_stat(Pde *pgdir, struct Page *pp, u_int p
        int count = 0;//统计满足条件的页表项的数量
        Pde *pde;
        Pte *pte;
        for (int i = 0; i < 1024; i++) {
                pde = pgdir + i;
                if(!(*pde & PTE_V)) { //当前页目录是否有效
                        continue;
                }

                for (int j = 0; j< 1024;j++ ){
                        pte = (Pte*)KADDR(PTE_ADDR(*pde))
                        if (!(*pte & PTE_V)) { ////当前页表
                                continue;
```

```
                        }
                        if (((perm_mask | (*pte))== (*pte
                && (((u_long)(page2pa(pp)))>>12) == (((u_
                                count++;
            /*该层if判断条件等价于
            (perm_mask & (*pte))== perm_mask
            (page2pa(pp) == PTE_ADDR(*pte))
            */
                }
    }
        return count;
}
```

lab2-extra

lab2-extra简单实现的**交换**技术，具体题目见文章Lab2-Extra-Swap题干。

题目题干中给出的思路和提示非常清晰，可以仔细品味，剩下的一些小细节，可以具体看下面答案中的注释。

答案

```
// Interface for 'Passive Swap Out'
struct Page *swap_alloc(Pde *pgdir, u_int asid) {
        // Step 1: Ensure free page
        if (LIST_EMPTY(&page_free_swapable_list)) {
                /* Your Code Here (1/3) */
                u_char  *disk_swap = disk_alloc();
                u_long da = (u_long)disk_swap;
                struct Page *p = pa2page(SWAP_PAGE_BASE);
                for (u_long i = 0; i < 1024; i++) { //改变
                        Pde *pde = pgdir + i;
                        if ((*pde) & PTE_V) {
                                for (u_long j = 0; j < 10
                                        Pte *pte = (Pte*)
                                        if (((*pte) & PTE_
                                                (*pte) =
                                                //上式作用
                                                (*pte) =
                                                tlb_inval
                                        }
                                }
                        }
                }
                memcpy((void *)da, (void *)page2kva(p), B
                //这里没有再删掉页控制块p对应的内容，是因为跳
                LIST_INSERT_HEAD(&page_free_swapable_list
        }
```

```
27
28          // Step 2: Get a free page and clear it
29          struct Page *pp = LIST_FIRST(&page_free_swapable_
30          LIST_REMOVE(pp, pp_link);
31          memset((void *)page2kva(pp), 0, BY2PG);
32
33          return pp;
34  }
35
36  // Interfaces for 'Active Swap In'
37  static int is_swapped(Pde *pgdir, u_long va) {
38          /* Your Code Here (2/3) */
39          Pde *pde = pgdir + PDX(va);
40          if (*pde & PTE_V) {
41                  Pte *pte = (Pte*)KADDR(PTE_ADDR(*pde)) + I
42                  if ((*pte & PTE_SWP) && !(*pte & PTE_V)) {
43                          return 1;
44                  }
45          }
46          return 0;
47  }
48
49  static void swap(Pde *pgdir, u_int asid, u_long va) {
50          /* Your Code Here (3/3) */
51          struct Page *pp = swap_alloc(pgdir, asid); //可用于
52          u_long da = PTE_ADDR(*((Pte*)KADDR(PTE_ADDR(*(pgd
53          memcpy((void *)page2kva(pp), (void *)da, BY2PG);
54
55          for (u_long i = 0; i < 1024; i++) { //所有页表项记录
56                  Pde *pde = pgdir + i;
57                  if (*pde & PTE_V) {
58                          for (u_long j = 0; j < 1024; j++)
59                                  Pte *pte = (Pte*)KADDR(PTE
60                                  if (!(*pte & PTE_V) && (*p
61                                          //以下三句话含义均可
62                                          (*pte) = ((page2pa
63                                          (*pte) = ((*pte) &
64                                          tlb_invalidate(asi
65                                  }
66                          }
67                  }
68          }
69          disk_free((u_char *)da);
70          return;
71  }
```

体会与感想

$lab2$ 好难啊，**页表机制**非常关键，**函数架构**比较复杂，无数的嵌套引用各个*函数*或*宏*，**指针**也是重中之重

操作系统的启动是一层套一层慢慢向上累加。在对页面操作时，我们用到了两种函数，一种是在内核刚刚启动的时候，这一部分内存通常用于存放内存控制块和进程控制块等数据结构，只能使用基础的 `alloc` ，一种则是在为用户创造环境之后按根据用户申请页面的需要进行操作。这体现了操作系统启动中每一步都是紧密相连的。

最重要的是就是学会阅读代码，以及尝试去使用现有的函数来完成新的需求，而不自己编写格外的函数或操作，在寻找现有的函数并使用的过程中就是在读懂代码。

debug方法

跳转

在命令行根目录输入：

```
1   ctags -R *    # 获得tags文件
```

在 Vim 中跳转：打开要查看的代码文件，将光标移动到 *函数名* 或 *结构体* 上，按下 `Ctrl+]` ，就可以跳转到 *函数* 或 *结构体* 的定义处。再按下 `Ctrl+O`（字母o）就可以返回跳转前的位置。

查找

在命令行查找：

```
1   grep -r page_init ./    # 在当前目录中查找所有文件（包括子目录）
```

在 Vim 中查找

1. 按 `/`
2. 输入搜索样式
3. 按 `Enter` 进行搜索
4. 按 `n` 搜索下一个匹配结果，或者 `N` 查找前面一个匹配结果

debugk.h法

1. 新建 `include/debugk.h` ：

```
#ifndef _DBGK_H_
#define _DBGK_H_
#include <printk.h>
#define DEBUGK // 可注释

#ifdef DEBUGK
#define DEBUGK(fmt, ...) do { printk("debugk::" fmt,
#else
#define DEBUGK(...)
#endif
#endif // !_DBGK_H_
```

2. 要想debug哪个 `.c` 文件就在文件前面加上 `#include <debugk.h>`

   在 `assert` 处想要输出的地方加上：

   ```
   DEBUGK("ckpt%d-待填入语句\n", 数字表示是加的第几个DEBUGK
   ```

3. 运行对应的 `make && make run`， 查看运行结果，如果卡在了哪里就能由 `debugk::ckpy` 看出来

4. 如果不行debug的内容输出影响对于整体程序的输出，可以注释掉 `include/debugk.h` 的下面这一行：

   ```
   /* #define DEBUGK // 可注释 */
   ```

GXemul法

```
make test lab=2_1 && make run
# 先笼统地看一下运行结果

make test lab=2_1  # 编译成相应的测试数据
make objdump       # 得到 target/mos.objdump 的反汇编文件
make dbg           # 进入调试模式
r, 0               # 查看CP0寄存器的值
在target/mos.objdump中查看CP0寄存器中epc地址处的语句，向上找-
在注释的位置帮助下，可以确定到底是哪条语句出了问题

breakpoint add [page_insert 或 0x80014560]    # 添加断点
c      # 运行到下一个断点，如果没有到断点，则说明根本不会执行至
s [n]  # 向后执行 n 条汇编指令
unassemble  # 导出某一个地址后续（或附近）的汇编指令序列
dump [curenv 或 0x804320e8]       # 导出某一个地址后续（或附
reg         # 导出所有寄存器的值
r, 0        # 导出CP0的寄存器值
tlbdump     # 导出TLB内容
trace       #
```