实验思考题

Thinking 2.1

- 在我们编写的 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?

  虚拟地址

- MIPS 汇编程序中 lw, sw 使用的是虚拟地址还是物理地址?

  物理地址

Thinking 2.2

- 请从可重用性的角度，阐述用宏来实现链表的好处。

  这样能将宏函数封装为一个整体，使其不再是一条条孤立的语句，而是一条语句，从而增加了宏函数的可移植性。

  同时， `do {/* ... */} while(0)` 语句的封装性更强，可以使内部的语句完全成为一个独立的代码块，防止了引用的时候上下文与宏内容产生关系。

  用的时候也不用管指针的实现，用就完事了。

- 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

```
#define SLIST_HEAD(name, type)                                          \
struct name {                                                           \
     struct type *slh_first; /* first element */                    \
}

#define SLIST_HEAD_INITIALIZER(head)                                    \
     { NULL }

#define SLIST_ENTRY(type)                                               \
struct {                                                                \
     struct type *sle_next;  /* next element */                     \
}
SLIST_INSERT_AFTER(slistelm, elm, field);
SLIST_INSERT_HEAD(head, elm, field);
SLIST_REMOVE_HEAD(head, field);
SLIST_REMOVE(head, elm, type, field);
```

```
/*
 * 单向链表没有 INSERT_BEFORE() 和 INSERT_TAIL()
 */
```

循环链表

```
#define CIRCLEQ_HEAD(name, type)                                        \
struct name {                                                           \
        struct type *cqh_first;         /* first element */             \
        struct type *cqh_last;          /* last element */              \
}

#define CIRCLEQ_HEAD_INITIALIZER(head)                                  \
        { (void *)&head, (void *)&head }

#define CIRCLEQ_ENTRY(type)                                             \
struct {                                                                \
        struct type *cqe_next;          /* next element */              \
        struct type *cqe_prev;          /* previous element */          \
}
CIRCLEQ_INSERT_AFTER(head, listelm, elm, field);
CIRCLEQ_INSERT_BEFORE(head, listelm, elm, field);
CIRCLEQ_INSERT_HEAD(head, elm, field);
CIRCLEQ_INSERT_TAIL(head, elm, field);
/*
 * 这里 INSERT_TAIL 时间复杂度 O(1)
 * MOS 的要遍历一遍，时间复杂度是 O(n)
 */
CIRCLEQ_REMOVE(head, elm, field);
```

Thinking 2.3

pmap.h 相关内容：

```
typedef LIST_ENTRY(Page) Page_LIST_entry_t;

struct Page {
Page_LIST_entry_t pp_link;    /* free list link */

// Ref is the count of pointers (usually in page table entries)
// to this page.  This only holds for pages allocated using
// page_alloc.  Pages allocated at boot time using pmap.c's "alloc"
// do not have valid reference count fields.

u_short pp_ref;
};
```

queue.h 相关内容：

```
#define LIST_HEAD(name, type)                                \
   struct name {                                             \
           struct type *lh_first; /* first element */ \
   }

#define LIST_ENTRY(type)
\
   struct {                                                                \
           struct type *le_next;  /* next element */                       \
           struct type **le_prev; /* address of previous next element */ \
   }
```

所以展开之后应该是：

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

所以你选择 C 项，并将其标在试卷上。

Thinking 2.4

请你寻找上述两个 boot_* 函数在何处被调用。

> `boot_pgdir_walk()` 在 `boot_map_segment()` 中调用。
> `boot_map_segment()` 在 `mips_vm_init()` 中调用。

Thinking 2.5

- 请阅读上面有关 R3000-TLB 的叙述，从虚拟内存的实现角度，阐述 ASID 的必要性

  > - ASID 可用来唯一标识进程，并为进程提供地址空间保护。在 MIPS 中，有两个地方会出现 ASID，每一个 TLB 表项会有一个 ASID，标识这个表项是属于哪一个进程的，CP0_EntryHi 中的 ASID 是当前进程的 ASID，所以进程对 TLB 的查询操作，即使 VPN 命中，但若该表项不是 global 且 ASID 与 CP0_EntryHi 的 ASID 不一致，则也视作 TLB 缺失。
  > - 除了提供地址空间保护外，ASID 允许 TLB 同时包含多个进程的条目。如果 TLB 不支持独立的 ASID，每次选择一个页表时（例如，上下文切换时），TLB 就必须被冲刷（flushed）或删除，以确保下一个进程不会使用错误的地址转换。所以有 ASID 就不用每次切换进程都要 flush 所有 TLB。

- 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

  > 通过设置具有特定 ASID 设置和 EntryLo G 位为零的 TLB 条目，这些条目将仅在 CPU 的 ASID 寄存器设置相同时匹配程序地址。 这允许软件同时映射多达 64 个不同的地址空间，而无需操作系统在上下文更改时清除 TLB。

Thinking 2.6

- tlb_invalidate() 和 tlb_out() 的调用关系是怎样的?

  > tlb_invalidate() 调用 tlb_out()

- 请用一句话概括 `tlb_invalidate()` 的作用

  > 给定特定进程的 context 这个函数可以让这个进程对应页表中的这个虚拟地址对应的 TLB 项失效。

- 逐行解释tlb_out中的汇编代码

```
1 #include <asm/regdef.h>
2 #include <asm/cp0regdef.h>
3 #include <asm/asm.h>
4
5 /* Exercise 2.10 */
6 LEAF(tlb_out) # 定义函数
```

```
//1: j 1b
nop
  /* 把原来 EntryHi 寄存器里的数据保存了下来
   *（因为一会儿要把我们的键值放进去，怕数据被覆盖了）
   */
  mfc0  k1,CP0_ENTRYHI

  mtc0  a0,CP0_ENTRYHI # 把键值移入 EntryHi
  nop
  tlbp # 查询 TLB 中的页表项，将查询的结果存入 Index 寄存器

  /* 连续出现4条 nop 指令的原因
   * 分别比较EntryHi, EntryLo0, EntryLo1, PageMask这四个寄存器的值,
   * 本操作系统的CPU没有针对tlbp指令作出相应的转发处理,
   * 因此只能通过暂停来解决指令间数据冒险,
   * 所以需要塞入4个nop阻塞，来保证读到的CP0_INDEX寄存器为最新值。
   */
  nop
  nop
  nop
  nop

  mfc0  k0, CP0_INDEX #把 Index 的内容移出来（Index 中保持着查询结果）
  bltz  k0, NOFOUND # 通过与 0 比较，判断有没有查到
  nop
  mtc0  zero, CP0_ENTRYHI # 将 EntryHi 清零
  mtc0  zero, CP0_ENTRYLO0 # 将 EntryLo 清零
  nop

  /* 向 TLB 中写入一个页表项信息
   * 将 EntryHi 和 EntryLo 的内容写入 index 对应的 tlb条目
   */
  tlbwi

NOFOUND: # 没查到
  mtc0  k1, CP0_ENTRYHI # 将原来的 EntryHi 给恢复了
  j ra # 返回
  nop
END(tlb_out)
```

Thinking 2.7

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，若记三级页表的基地址为 PT_{base}，请你计算：

- 三级页表页目录的基地址

$$PD_{base} = PT_{base}|(PT_{base} >> 9)|(PT_{base} >> 18)$$

- 映射到页目录自身的页目录项（自映射）

$$PDE_{self-mapping} = PT_{base}|(PT_{base} >> 9)|(PT_{base} >> 18)|(PT_{base} >> 27)$$

Thinking 2.8

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 体系结构中的内存管理机制：

1. 分段机制

分段是一种朴素的内存管理机制，它将内存划分成以起始地址 base 和长度 limit 描述的块，这些内存块就称为段。段可以与程序最基本的元素联系起来。例如程序可以简单地分为代码段、数据段和栈，段机制中就有对应的代码段、数据段和栈段。

分段机制由 4 个基本部分构成：逻辑地址、段选择寄存器、段描述符和段描述符表。其核心思想是：使用段描述符描述段的基地址、长度以及各种属性。当程序使用逻辑地址访问内存的某个部分时，CPU 通过逻辑地址中的段选择符索引段描述符表以得到该内存对应的段描述符。并检验程序的访问是否合法，如合法，根据段描述符中的基地址将逻辑地址转换为线性地址。

2. 分页机制

X86 的分页机制是更加粒度化的内存管理机制，与分段机制将内存划分成以基地址和长度描述的多个段进行管理不同，分页机制是用粒度化的单位页来管理线性地址空间和物理地址空间。X86 架构下一个典型的页大小是 4KB，则一个 4GB的虚拟地址空间可以划分成 1024x1024 个页面。物理地址空间划分同理。x86 架构允许大于 4KB 的页面大小，这里只介绍 4KB 的页面管理机制。

分页机制让现代操作系统中的虚拟内存机制成为可能，感谢这种机制，一个页面可以存在于物理内存中，也可以存放在磁盘的交换区域中。程序可以使用比物理内存更大的内存区域。分页机制的核心思想是通过页表将线性地址转换成为物理地址，并配合旁路转换缓冲区来加速地址转换过程。操作系统在启动过程中，通过将 CR0 寄存器的 PG 位置 1 来启动分页机制。分页机制主要由页表、CR3 寄存器和 TLB 三个部件构成。

页表 Page Table 是用于将线性地址转换成物理地址的主要数据结构。一个地址对齐到页边界后的值称为页帧号，它实际是该地址所在页面的基地址。线性地址对应的也帧号即虚拟页帧号 VFN，物理地址对应的页帧号即物理地址页帧号 PFN 或机器页帧号。故也可以认为，页表是存储 VFN 到 PFN 映射的数据结构。4KB 大小的页面使用两级页表。

页目录项 PageDirectoryEntry 包含页表的物理地址。页目录项存放在页目录中。CPU 使用线性地址的 22–31 位索引页目录，以获得该线性地址对应的页目录项。每个页目录项为 4B 大小，故页目录占用一个 4KB 大小的物理页面，共包含 1024 的页目录项。

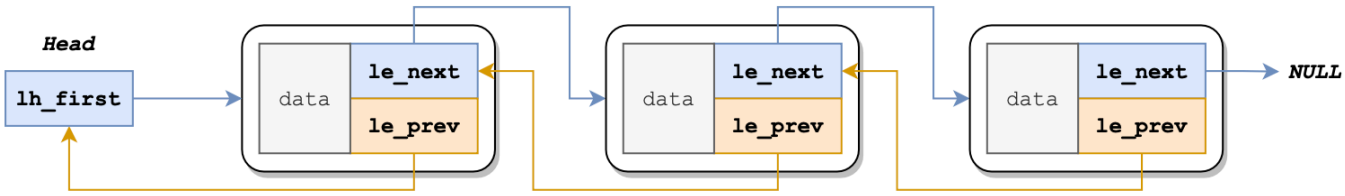
页表项 PageTableEntry 页表项包含该线性地址对应的 PFN。页表项存放在页表中。CPU 使用 12–21 位索引页表，获得该线性地址对应的页表项。通过将线性地址的 0–11 位偏移量和基地址相加，就可以得到线性地址对应的物理地址。页表项为 4B 大小，故页表包含 1024 个页表项，占用一个 4KB 页面。

MIPS 与 X86 的 TLB 差别在于对 TLB 不命中时的处理上：

- MIPS 会触发TLB Refill 异常，内核的 tlb_refill_handler 会以 pgd_current 为当前进程的 PGD 基址，索引获得转换失败的虚址对应的 PTE，并将其填入 TLB，完了CPU 把刚刚转换失败的虚地址再走一下 TLB 就可以了。
- X86 在 TLB 不命中时，是由硬件 MMU 以 CR3 为当前进程的 PGD 基址，索引获得 PFN 后，直接输出 PA。同时 MMU 会填充 TLB 以加快下次转换的速度。
- 转换失败的虚址，MIPS 使用 BadVAddr 寄存器存放，X86 使用 CR2 存放。

实验难点图示

链表的操作，宏函数编写



主要要注意的就是
每一个结构里面的 le_prev 指向的是前一个结构的 le_next，而不是上个结构本身。

页面与地址的转化

在本次实验中涉及到许多的页面与地址的转化，其中用到许多已经定义的函数：

page2pa：得到某个page结构体的物理地址

```
/* Get the physical address of Page 'pp'.
 */
static inline u_long
page2pa(struct Page *pp) {
    return page2ppn(pp) << PGSHIFT;
}
```

pa2page：得到某个物理地址所对应的Page结构体

```
/* Get the Page struct whose physical address is 'pa'.
 */
static inline struct Page *
pa2page(u_long pa)
{
  if (PPN(pa) >= npage) {
    panic("pa2page called with invalid pa: %x", pa);
  }

  return &pages[PPN(pa)];
}
```

page2kva：得到某个Page结构体的内核虚拟地址

```
/* Get the kernel virtual address of Page 'pp'.
 */
static inline u_long
page2kva(struct Page *pp) {
    return KADDR(page2pa(pp));
}
```

PPN：得到某个虚拟地址的页号

```
#define PPN(va)   (((u_long)(va))>>12)
```

PADDR：将某个内核虚拟地址转化为物理地址

```
// translates from kernel virtual address to physical address.
#define PADDR(kva)                                              \
    ({                                                          \
        u_long a = (u_long)(kva);                               \
        if (a < ULIM)                                           \
            panic("PADDR called with invalid kva %08lx", a); \
        a - ULIM;                                               \
    })
```

KADDR：将某个物理地址转化为内核虚拟地址

```
// translates from physical address to kernel virtual address.
#define KADDR(pa)                                                   \
    ({                                                              \
        u_long ppn = PPN(pa);                                       \
        if (ppn >= npage)                                           \
            panic("KADDR called with invalid pa %08lx", (u_long)pa); \
        (pa) + ULIM;                                                \
    })
```

本次实验的前半部分涉及了许多对这类函数的应用，熟练掌握这类函数实现各类地址查询是本次实验的一大难点。

体会与感想

这次的实验可以说让我感受到了很大的跨度和极高的挑战度。结合理论课上学习到的知识，我对这些知识做了进一步树立，用于应对这一次的实验内容，并通过补全代码进一步强化我对 cache，MMU，TLB等概念的理解与掌握。实验中遇到了很多的问题，大多表现为看到一个需求不知道怎么下手。

虽然 lab2 的所有代码补全基本全部集中在 pamp.c 这一个文件中，但是在地址转化等操作中，所用到的函数和宏函数却遍布于实验操作系统的每一个文件中。我在课下实验用的方法是将代码下载到实体机，在自己的电脑上进行补全后再 cv 到虚拟机中，中间大量利用了编辑器强大的搜索功能。但即使是这样，我仍然在寻找函数时遇到了很多困难。

还有一个问题，那就是我发现我的 C 语言基础不牢靠，在面对 lab2 众多的指针时有些吃不消。看来 C 语言还是得重修。

残留难点

对我而言，这次实验中残留的难点主要是有关物理地址与虚拟地址的区别，还是不太清楚是怎么分辨和转化，没有注释的话还是不太能看得出来。