lab2 实验报告

一、思考题

Thinking 2.1 请思考 cache 用虚拟地址来查询的可能性，并且给出这种方式对访存带来的好处和坏处。另外，你能否能根据前一个问题的解答来得出用物理地址来查询的优势?

A: cache 用虚拟地址查询存在问题：1. 因为不同的进程的虚地址可能是一样的，因此用虚地址进行 cache 查询可能会有冲突。2. IO 设备是直接物理映射，没有从物理地址转换到虚拟地址的步骤。3. 现代操作系统有些多个虚拟页面对应同一个物理空间，同样可能会出现读写不一致的情况。
因此，物理地址查询可以避免虚拟地址和物理地址没有一一对应时的读写不一致。以及节省 IO 设备转换成虚拟地址的额外开销。

Thinking 2.2 请查阅相关资料，针对我们提出的疑问，给出一个上述流程的优化版本，新的版本需要有更快的访存效率。(提示:考虑并行执行某些步骤)

A: 虚拟地址和物理地址相结合。即虚拟地址和物理地址的低位通常相同的，不同的是高位映射。在第一个时钟周期先对 TLB 和 cache 同时进行查找，并取出相应数据（如果命中），并且 TLB 转换物理地址的 TAG。第二个时钟周期根据转换得到的物理地址的 TAG 判断 cache 数据是否采纳。

Thinking 2.3 在我们的实验中，有许多对虚拟地址或者物理地址操作的宏函数(详见 include/mmu.h),那么我们在调用这些宏的时候需要弄清楚需要操作的地址是物理地址还是虚拟地址，阅读下面的代码，指出 x 是一个物理地址还是虚拟地址。

A: 虚拟地址。

Thinking 2.4 我们注意到我们把宏函数的函数体写成了 do { /* ... */ } while(0) 的形式，而不是仅仅写成形如 { /* ... */ } 的语句块，这样的写法好处是什么?

A: 因为，如果在宏函数中有不止一句代码，则在后文调用这个宏函数时，可能会发生逻辑上的错误。比如在 if 语句后调用这个宏，则只有第一句遵循 if 条件判断，其余都必然执行。或者在宏定义的时候就用大括号括起来，那容易发生语法错误，即多使用了一个分号。所以宏函数加 do{…}while(0)其实并不是额外加了

什么功能，只是为了保持宏函数实现的时候语意和我们所写的代码一致，防止大括号以及分号的干扰。

Thinking 2.5 注意，我们定义的 Page 结构体只是一个信息的载体，它只代表了相应物理内存页的信息，它本身并不是物理内存页。那我们的物理内存页究竟在哪 呢?Page 结构体又是通过怎样的方式找到它代表的物理内存页的地址呢?请你阅读 include/pmap.h 与 mm/pmap.c 中相关代码，给出你的想法。
Page 结构体存储时是连续页面，因此可以首先计算页号，再通过移位获取物理地址。

Thinking 2.6
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(请注意指针)。
C。由

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

可知，name 即为 Page_list，因此，*lh_first;在 Page_list 结构体内，索引第一个 Page。这是最外面一层结构。
再往里的 struct type，是 Page 结构体，由

```
struct Page {
    Page_LIST_entry_t pp_link;/* free list link */
    u_short pp_ref;
};
```

可知，pp_ref 在这一层结构体中，再往里是 Page_LIST_entry_t 类型，由

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

可知，这一层的结构体含有指向 Page 类型的两个指针。
因此，一个页表结构体里有指向首页的指针，页结构体里有页的前后 link 结构体和该页被引用次数的 counter，页的前后 link 结构体内含有两个指向页的指针。因此，正确的展开结构应为 C。

Thinking 2.7

在 mmu.h 中定义了|bzero(void *b, size_t)|这样一个函数,请你思考，此处的 b 指针是一个物理地址， 还是一个虚拟地址呢？

是虚拟地址。b 传入的是 alloced_mem，是被分配的虚拟地址。

Thinking 2.8

了解了二级页表页目录自映射的原理之后，我们知道，Win2k 内核的虚存管理也是采用了二级页表的形式，其页表所占的 4M 空间对应的虚存起始地址为 0xC0000000，那么，它的页目录的起始地址是多少呢？

由于地址 0xC000_0000 对应的是第(0xC000_0000>>12)个页表项，即第 0xC0000 个页表项。这个页表项也就是第一个页目录项。一个页表项 32 位，占用 4B 内存，因此相对于页表起始位置的偏移是 (0xC000_0000>>12)*4=0x30_0000。因此，页目录的需存地址是 0xC000_0000 + 0x30_0000 = 0xC030_0000。

Thinking 2.9

思考一下 tlb_out 汇编函数，结合代码阐述一下跳转到 NOFOUND 的流程？从 MIPS 手册中查找 tlbp 和 tlbwi 指令，明确其用途，并解释为何第 10 行处指令后有 4 条 nop 指令。

流程：寄存器加载了 TLB 快表项的地址，和 entry HI 寄存器内容匹配。若没有 TLB 条目匹配，则 index 寄存器的高位置位。取出 index 寄存器的值，如果小于 0，说明高位为 1，表示没有 TLB 条目匹配，于是跳转到 NOFOUND 位置。

tlbp 指令：index 寄存器加载了 TLB 快表项的地址，和 entry HI 寄存器内容匹配。若没有 TLB 条目匹配，则 index 寄存器的高位置位。取出 index 寄存器的值，如果小于 0，说明高位为 1，表示没有 TLB 条目匹配，于是跳转到 NOFOUND 位置。

tlbwi： TLB 表项写入，由 EntryHi, EntryLo0, EntryLo1, and PageMask 寄存器共同决定内容。

4 条 nop 指令：因为 tlbp 指令需要执行多个周期。找 TLB 条目–》是否匹配–》有则置位。

Thinking 2.10

显然，运行后结果与我们预期的不符，va 值为 0x88888，相应的 pa 中的值为 0。这说明我们的代码中存在问题，请你仔细思考我们的访存模型，指出问题所在。

Lab2 中的 TLB 没有异常处理机制。在 TLB update 时，调用了 tlb_out 这个汇编函数，会将相应的 va 页从 TLB 里删除。当 TLB 缺失时，引起异常而并没有进行缺页处理。因此我的程序就陷入了死循环。而当 va 的值为 0x7f40_0000 或者 0x7f80_0000 等这类值时，就不会出现异常，而是能获得正确的 pa 值。

Thinking 2.11

在 X86 体系结构下的操作系统，有一个特殊的寄存器 CR4，在其中有一个 PSE 位，当该位设为 1 时将开启 4MB 大物理页面模式，请查阅相关资料，说明当 PSE 开启时的页表组织形式与我们当前的页表组织形式的区别。

使用 PSE 技术的话，页目录中的表项中有一个 page size 标记（PS），当 PS = 1 时，这个页目录表项就指向了一个 4MB 的大物理页面，而不是像原来那样指向一个页表。同时，实际上只是将 32 位地址的后 22 位全部置零，这样就可以指向一个 4MB 对齐的物理页面。

二、实验难点图示

难点在于理解虚拟内存初始化的整个过程。以及指针相关知识遗忘率较大。

我的理解是这样的：

建立内存系统的步骤主要有三步：

```
mips_detect_memory();   // 设定容量初始值
mips_vm_init();         // 建立二级页表系统
page_init();            // 初始化页面和空闲页面链表
```

具体操作如下。

1. mips_detect_memory()

```
void mips_detect_memory()
{
    /* Step 1: Initialize basemem.
     * (When use real computer, CMOS tells us how many kilobytes there are). */
     maxpa = 0x04000000; // ps. maxpa == basemem == 64MB
     basemem = 64 * 1024 * 1024; // B  (64MB)
     // Step 2: Calculate corresponding npage value.
     npage = basemem/(1024*4); // (4 KB/page)
     extmem = 0;

    printf("Physical memory: %dK available, ", (int)(maxpa / 1024));
    printf("base = %dK, extended = %dK\n", (int)(basemem / 1024),
           (int)(extmem / 1024));
}
```

设定初始值。我们可以自由调度从 end 到 Physics Memory Max 这一部分的虚存地址。此段内存大小是 64MB。可以被划分成 npage 个 4KB 大小的页面。

2. mips_vm_init()

其次是创建二级页表系统。

第一步，分配页目录。

```
/* Step 1: Allocate a page for page directory(first level page table). */
pgdir = alloc(BY2PG, BY2PG, 1); // 分配页目录
printf("to memory %x for struct page directory.\n", freemem);
mCONTEXT = (int)pgdir;
```

用 alloc 函数，分配 BY2PG 大小的一页。函数返回被分配的地址。在这里是首地址 0x8040_0000。

第二步，分配存储页结构体的空间。

```
pages = (struct Page *)alloc(npage * sizeof(struct Page), BY2PG, 1);
printf("to memory %x for struct Pages.\n", freemem);
n = ROUND(npage * sizeof(struct Page), BY2PG);
boot_map_segment(pgdir, UPAGES, n, PADDR(pages), PTE_R);
```

用途是存储结构体指针等。空间大小是 npage*Page 结构体大小。
*pages 是指向这些页的首地址的指针，因此*pages 得到的是页，pages 存储的是地址。

n 计算了存储这些结构体所需的空间页数，并圆整。由于所有页表总空间为 64MB，每页 4KB，故总共有 npage=64MB/4KB 页，即 16384 页，Page 结构体的大小为 12Bytes，因此需要空间 196608B，所以存储这些结构体需要 196608B/4KB=48 页，即 n=48。
boot_map_segment()函数，将相应物理页面的地址填入对应的虚拟地址的页表项中。相当于将物理地址和虚拟地址一一映射。由 boot_pgdir_walk()这个函数首先申请一个新页表，返回页表项入口的虚拟地址，然后将物理页框号和有效位赋值到这个地址上。这个过程中，pgdir_entryp 的值始终为 0x804007f8（7f8 的原因应该是因为 mmu.h 里定义的 PAGES 的地址是从 0x7f80_0000 开始吧），表示这 48 页对应的页目录都是这一项，二级页表的虚地址的值从 0x8043_1000 到 0x8043_10b8，每次+4，二级页表中存储的物理地址，从 0x401000 到 0x430000。

第三步，为进程分配空间。

```
envs = (struct Env *)alloc(NENV * sizeof(struct Env), BY2PG, 1);
n = ROUND(NENV * sizeof(struct Env), BY2PG);
boot_map_segment(pgdir, UENVS, n, PADDR(envs), PTE_R);
```

这一步和第二步差不多，只是在分配地址空间的时候，对应的 pgdir_entryp 是 0x8040_07f4（因为在 mmu.h 里分配 ENVS 的空间是从 0x7f40_0000 开始？），分配的页数是 54 页。二级页表的地址从 0x8046_8000 到 0x8046_80d4，对应物理页框号从 0x432000 到 0x467000。

中间还有一页物理地址为 0x431000 的物理页框，被用于存放指向 PAGES 的页表，而后面还有一页物理地址为 0x468000 的页框，被用于存放指向 ENVS 的页表。而这两页的空间各是 0x1000 是 4KB，但是实际使用的空间仅为 0xb8 和 0xd4。但是因为圆整函数 ROUND，所以在下一次分配空间的时候会以 0x1000 对齐来分配。

3. page_init()

在这一步中，将之前所有申请的 page 的 ref 初始化，page 的结构体由两部分组成，一个是指针域（其中包含两个指针*le_next 和**le_prev），这个在之前已经完成，然后就是 ref 的初始化。以及未使用的空闲页面连接到 page_free_list 中。
这样，二级页表系统的初始化工作就完成了。
至此，已经构建好的二级页表系统的地址映射大致如下：

内容	4KB 页目录	192KB 的 Pages 结构体	4KB 的 Pages 的页表	216KB 的 env 结构体	4KB 的 env 的页表
虚地址	0x8040_0000	页表项：0x8040_1000~0x8043_0000 结构体：0x7f800000~0x7fc00000	0x8043_1000~0x8043_10b8	页表项：0x8043_2000~0x8046_7000 结构体：0x7f400000~0x7f800000	0x8046_8000~0x8046_80d4
实地址	0x40_0000	0x40_1000~0x43_0000	0x43_1000	0x43_2000~0x46_7000	0x46_8000

关于 page_check()

这个函数首先把剩余 page_free_list 暂存到 fl 中。然后用 LIST_INIT()初始化，所以此时的 page_free_list 是空的。相当于刚开始我们把空闲物理页放到 page_free_list 都白干。
接着开始 page_insert()。但是由于没有空闲物理页，会先报 NO_MEM，所以必须 free 一些页。之后可以成功 insert 一些页。并且它的物理地址是从最高位往下。即 0x3fff000 是新创建的页表的物理页框号，对应的二级页表的首地址是 0x83ff_f000，0x3ffe000 是新的一页的物理页框号，二级页表项入口是 0x83ff_f004，接着是 0x3ffd000，以此类推。

三、体会与感想

Lab2 真的很难理解，刚开始的时候只是按照 step 的提示把每一个函数填充好了，这些提示真的很简略。同样，难点不在于函数本身的填写，而是在于理解虚拟地址和物理地址的一一映射关系，是如何实现的，以及为什么要这样做和为什么

可以这样做。直到自己加了很多 print 之后才通过打印输出的地址和他们的值，联系他们经历的函数才慢慢明白整个过程。页表、页目录也需要物理页去存那些指针和值，映射关系通过指针的指向和对应存储的物理页框号来实现，这样就能将虚实地址映射起来。

我觉得 lab2 就像一个解谜故事，线索非常少，但是事实就在那里，代码也就那么几行，而正确输出就在那里。所以再怎么难懂，终究还是可以通过蛛丝马迹和推理猜测学会这些。虽然不一定对，就像福尔摩斯也不一定能推理正确谜题的每一个真相。

四、指导书反馈

指导书和代码里的英文太 Chinglish 了，有很大的误导性。以及内容太少。我觉得这些代码全都自己看懂有点多，或许可以加一些提示和指导。比如：

1. 建议先看 xx 函数，再看 xx 函数。
2. 函数执行顺序为 blabla。
3. 提示：在 page_check()函数中有将 page_free_list 初始化的操作，并不是我们已经把空闲页都用完了。
4. 我们先初始化了 PAGES，再初始化了 ENVS……然后二级页表的虚存管理系统就建立起来了。

等等诸如此类的提示性 hint，这样可以帮助我们事先建立一个大概的框架，明白已经做了什么，我们要做什么。现在的指导书更像是按照知识点来介绍，而缺少一个清晰的路线。

五、 遗留问题

1. mips_vm_init()里

```
pages = (struct Page *)alloc(npage * sizeof(struct Page), BY2PG, 1);
```

为什么这里 pages 是 Page 类型的指针？我的理解是它存储的是物理页框的地址。

2. 物理页框号是从 0x400_0000 到 0x3ff_ff00 吗？所以说还没有 64MB 这么大？
3. Thinking 2.10 不是很懂。应该说 lab2 里的 TLB 相关都不太懂。以及，调用 va2pa 将 va 转换成 pa，并不总能得到我想要的 pa 值，而经常是 fffffff。