实　　验　　报　　告

学　　号	21030031009	姓　　名	惠欣宇	专业班级	计算机 4 班
课程名称	**操作系统**			学期	**2023 年秋季学期**
任课教师	窦金凤	完成日期	2023. 12. 15	上机课时间	周五 3-6
实 验 名 称	内存管理				

Exercise 部分：

Exercise 2.1

> Exercise 2.1 在阅读 queue.h 文件之后，相信你对宏函数的巧妙之处有了更深的体会。请完成 queue.h 中的LIST_INSERT_AFTER函数和LIST_INSERT_TAIL函数。 ■

定义一个宏 **LIST_INSERT_AFTER**，用于在链表中的某个元素之后插入新的元素，其中每个元素包含一个指向下一个元素的指针。
LIST_INSERT_AFTER(listelm, elm, field): 宏接受三个参数。
listelm: 链表中的某个元素，表示待插入位置的元素。
elm: 要插入的新元素。
field: 表示链表中用于链接下一个元素的字段。
实现逻辑：
将新元素 elm 的 LIST_NEXT 指针设置为待插入位置元素 listelm 的下一个元素。
如果待插入位置元素 listelm 有下一个元素，更新下一个元素的 le_prev 指针，使其指向新元素 elm 的 LIST_NEXT 指针。
将待插入位置元素 listelm 的 LIST_NEXT 指针设置为新元素 elm，同时将新元素 elm 的 le_prev 指针设置为待插入位置元素 listelm 的 LIST_NEXT 指针的地址。
能够正确地更新链表的指针关系，保持链表的连续性。
完成结果如图 1 所示。

```
112 #define LIST_INSERT_AFTER(listelm, elm, field) do {                    \
113             LIST_NEXT((elm),field) = LIST_NEXT((listelm),field);          \
114             if((LIST_NEXT((listelm),field))){                             \
115                 (LIST_NEXT((listelm),field))->field.le_prev = &LIST_NEXT((elm),field);\
116             } \
117                 LIST_NEXT((listelm),field) = (elm);\
118                 (elm)->field.le_prev = &LIST_NEXT((listelm),field);\
119                 \
120     } while (0)
```

图 1　LIST_INSERT_AFTER 的完成结果

定义一个宏 **LIST_INSERT_TAIL**，用于将一个元素插入到链表的尾部。
LIST_INSERT_TAIL(head, elm, field): 宏接受三个参数。
head: 链表的头部，表示链表的起始位置。
elm: 要插入的新元素。
field: 表示链表中用于链接下一个元素的字段。
实现逻辑：
如果链表不为空（LIST_FIRST((head)) != NULL），则执行以下步骤：
将新元素 elm 的 LIST_NEXT 指针设置为链表的头部。
遍历链表，找到最后一个元素，将新元素 elm 插入到链表的尾部。
将新元素 elm 的 le_prev 指针设置为最后一个元素的 LIST_NEXT 指针的地址。
将新元素 elm 的 LIST_NEXT 指针设置为 NULL，表示它是最后一个元素。

如果链表为空，调用 LIST_INSERT_HEAD 将新元素插入到链表的头部。
完成结果如图 2 所示。

```
#define LIST_INSERT_TAIL(head, elm, field) do{                                  \
    if(LIST_FIRST((head))!= NULL){                                              \
        LIST_NEXT((elm),field) = LIST_FIRST((head));                            \
        while(LIST_NEXT((LIST_NEXT((elm),field)),field) != NULL){               \
            LIST_NEXT((elm),field) = LIST_NEXT(LIST_NEXT((elm),field),field);\
        }   \
        LIST_NEXT(LIST_NEXT((elm),field),field ) = (elm);\
        (elm)->field.le_prev = &LIST_NEXT(LIST_NEXT((elm),field),field); \
        LIST_NEXT((elm),field) = NULL;\
    } else { \
        LIST_INSERT_HEAD((head),(elm),field);\
    }\
} while (0)
```

图 2　LIST_INSERT_TAIL 的完成结果

Exercise 2.2

> **Exercise 2.2** 我们需要在 mips_detect_memory() 函数中初始化这几个全局变量，以确定内核可用的物理内存的大小和范围。根据代码注释中的提示，完成 mips_detect_memory() 函数。 ■

函数 mips_detect_memory 是在 MIPS 架构下检测系统内存的。将物理内存的最大地址设定为 0x4000000。将基础内存大小初始化为 0x4000000。计算对应的页表项数目 npage 为 0x4000。将扩展内存大小初始化为 0。
输出一些有关内存的信息，包括物理内存大小、基础内存和扩展内存的大小，使用 printf 函数打印到控制台。
完成结果如图 3 所示。

```
void mips_detect_memory()
{
    /* Step 1: Initialize basemem.
     * (When use real computer, CMOS tells us how many kilobytes there are). */
    maxpa = 0x4000000;
    basemem = 0x4000000;
    npage = 0x4000;
    extmem = 0;
    // Step 2: Calculate corresponding npage value.
    printf("Physical memory: %dK available, ", (int)(maxpa / 1024));
    printf("base = %dK, extended = %dK\n", (int)(basemem / 1024),
           (int)(extmem / 1024));
}
```

图 3　mips_detect_memory 的完成结果

Exercise 2.3

> **Exercise 2.3** 完成 page_init 函数，使用 include/queue.h 中定义的宏函数将未分配的物理页加入到空闲链表 page_free_list 中去。思考如何区分已分配的内存块和未分配的内存块，并注意内核可用的物理内存上限。 ■

函数 page_init 用于初始化物理页面管理。
初始化 page_free_list：通过 LIST_INIT(&page_free_list)，初始化一个链表 page_free_list，用于管理空闲的物理内存页。

对齐 freemem 到页面边界：使用 ROUND(freemem, BY2PG)，将 freemem 向上对齐到页面大小的倍数。这可能是为了确保 freemem 在一个物理页面的边界上。
标记低于 freemem 的内存为已使用：通过循环遍历 pages 数组，将 pp_ref 字段设置为 1，以表示这些页面已被使用。
标记其余内存为可用： 通过循环遍历从 freemem 开始的剩余内存，将 pp_ref 字段设置为 0。对于非栈区的页面，将其添加到 page_free_list 链表中，表示这些页面是可用的。
完成结果如图 4 所示。

```
191 void page_init(void)
192 {
193     struct Page *now;
194     struct Page *last;
195     /* Step 1: Initialize page_free_list. */
196     LIST_INIT(&page_free_list);
197     /* Step 2: Align `freemem` up to multiple of BY2PG. */
198     freemem = ROUND(freemem, BY2PG);
199     /* Step 3: Mark all memory blow `freemem` as used(set `pp_ref`* filed to 1) */
200     for(now = pages; page2kva(now) < freemem; now++ ){
201         now->pp_ref = 1;
202     }
203     /* Step 4: Mark the other memory as free. */
204     for(now = &pages[PPN(PADDR(freemem))];page2ppn(now) < npage; now++){
205         if(page2ppn(now) != PPN(TIMESTACK - 1) ){
206             now->pp_ref = 0;
207             LIST_INSERT_HEAD(&page_free_list, now, pp_link);
208         }else{
209             now->pp_ref = 0;
210         }
211     }
212 }
```

图 4　page_init 的完成结果

Exercise 2.4

> **Exercise 2.4** 完成 mm/pmap.c 中的 page_alloc 和 page_free 函数，基于空闲内存链表 page_free_list ，以页为单位进行物理内存的管理。并在 init/init.c 的函数 mips_init 中注释掉 page_check();。此时运行结果如下。 ■

函数 page_alloc 用于从物理页面管理中分配页面。
获取空闲页面： 通过检查空闲页面链表 page_free_list 是否为空（LIST_EMPTY(&page_free_list)），如果为空，则表示没有可用的内存页，函数返回错误码 -E_NO_MEM 表示内存不足。
从空闲页面链表中取出一个页面： 通过 LIST_FIRST(&page_free_list) 获取空闲链表的第一个页面，然后使用 LIST_REMOVE 从链表中移除这个页面。
初始化取出的页面： 使用 bzero 函数将取出的页面内容清零，确保页面中的数据为空。
返回分配的页面： 将分配的页面的指针赋值给 pp，这是作为输出参数传递的。
返回成功状态码： 函数返回状态码 0，表示成功分配内存页。
完成结果如图 5 所示。

```c
int page_alloc(struct Page **pp)
{
    struct Page *ppage_temp;
    struct Page *tmp;
    /* Step 1: Get a page from free memory. If fail, return the error code.*/
    if(LIST_EMPTY(&page_free_list)){
        return -E_NO_MEM;
    }
    tmp = LIST_FIRST(&page_free_list);
    LIST_REMOVE(tmp,pp_link);
    /* Step 2: Initialize this page.
     * Hint: use `bzero`. */
    bzero((void *)page2kva(tmp),BY2PG);
    *pp = tmp;
    return 0;
}
```

图 5　page_alloc 的完成结果

函数 page_free 用于释放物理页面。
检查页面是否有虚拟地址引用：如果 pp 结构体中的 pp_ref 字段为 0，表示没有虚拟地址引用这个页面，那么执行以下操作：使用 LIST_INSERT_HEAD 将页面插入到空闲页面链表 page_free_list 的头部。返回，不再执行后续的步骤。
检查 pp_ref 的值：如果 pp_ref 大于 0，表示仍有虚拟地址引用该页面，则直接返回，不进行释放操作。
处理 pp_ref 小于 0 的情况： 如果 pp_ref 小于 0，则发生了错误，触发内核恐慌（panic），并输出错误信息，中止程序执行。
完成结果如图 6 所示。

```c
void page_free(struct Page *pp)
{
    /* Step 1: If there's still virtual address referring to this page, do nothing. */
    if(pp->pp_ref == 0){
        LIST_INSERT_HEAD(&page_free_list, pp, pp_link);
        return;
    } else if (pp -> pp_ref > 0)
        return;
    /* Step 2: If the `pp_ref` reaches 0, mark this page as free and return. */
    panic("cgh:pp->pp_ref is less than zero\n");
}
```

图 6　page_ free 的完成结果

Exercise 2.5

> **Exercise 2.5** 完成 mm/pmap.c 中的 boot_pgdir_walk 和 pgdir_walk 函数，实现虚拟地址到物理地址的转换以及创建页表的功能。 ■

函数 boot_pgdir_walk 用于实现页表的查找和创建。
获取页目录项指针：通过 pgdir + PDX(va) 获取虚拟地址 va 对应的页目录项在页目录数组 pgdir 中的指针。
检查页目录项是否存在：通过 *pgdir_entry & PTE_V 检查页目录项是否存在（是否合法）。
如果不存在（为 0），并且需要创建（create 参数为真），则执行以下步骤：
分配一个新的页表，将其物理地址设置到页目录项中。
将页目录项设置为合法（PTE_V）和可读（PTE_R）。
如果不存在且不需要创建，则返回 0，表示无法继续查找。
获取页表的指针：通过 KADDR(PTE_ADDR(*pgdir_entry)) 获取页表的虚拟地址，即页表的起始地址。
获取页表项指针：通过 pgtable + PTX(va) 获取虚拟地址 va 对应的页表项在页表中的指针。
返回页表项指针：将获取到的页表项指针返回，表示查找或创建成功。
完成结果如图 7 所示。

```
88  static Pte *boot_pgdir_walk(Pde *pgdir, u_long va, int create)
89  {
90      Pde *pgdir_entry;
91      Pte *pgtable, *pgtable_entry;
92      pgdir_entry = pgdir + PDX(va);
93      if((*pgdir_entry & PTE_V) == 0 ){
94          if(create){
95              *pgdir_entry = PADDR(alloc(BY2PG,BY2PG,1));
96              *pgdir_entry = (*pgdir_entry) | PTE_V | PTE_R;
97
98          }else return 0;
99      }
100     pgtable = (Pte *)(KADDR(PTE_ADDR(*pgdir_entry)));
101     pgtable_entry = pgtable + PTX(va);
102     return pgtable_entry;
103 }
```

图 7　boot_pgdir_walk 的完成结果

Exercise 2.6

> **Exercise 2.6** 实现 mm/pmap.c 中的 boot_map_segment 函数，实现将制定的物理内存与虚拟内存建立起映射的功能。

函数 boot_map_segment 用于将一个物理内存段映射到虚拟地址空间。
循环映射页表项： 通过 for 循环，以页为单位遍历指定大小的虚拟地址范围。
获取页表项指针： 通过调用 boot_pgdir_walk 函数获取虚拟地址 va + i 对应的页表项指针，如果需要创建页表的话，函数内部会进行创建。
设置页表项： 将页表项设置为物理地址 pa + i（在物理地址的基础上加上偏移 i）、权限 perm（如读、写、执行等权限）以及标志位 PTE_V（表示该页表项有效）。
完成结果如图 8 所示。

```
113 void boot_map_segment(Pde *pgdir, u_long va, u_long size, u_long pa, int perm)
114 {
115     int i, va_temp;
116     Pte *pgtable_entry;
117     for(i=0,size = ROUND(size, BY2PG); i < size; i += BY2PG){
118         pgtable_entry = boot_pgdir_walk(pgdir, va + i, 1 );
119         *pgtable_entry = (pa + i) | perm | PTE_V;
120
121     }
122 }
```

图 8　boot_map_segment 的完成结果

Exercise 2.7

> **Exercise 2.7** 完成 mm/pmap.c 中的 page_insert 函数。

函数 page_insert 将一个物理页面映射到指定的虚拟地址，并设置相应的权限。
获取页表项指针： 通过调用 pgdir_walk 函数获取虚拟地址 va 对应的页表项指针，并将结果保存在 pgtable_entry 中。
检查页表项状态：
如果 pgtable_entry 不为空且页表项已存在（(*pgtable_entry & PTE_V) != 0）：
检查已存在的页表项是否对应于输入的物理页面 pp。
如果不对应，则调用 page_remove 函数移除已存在的映射。
如果对应，则更新页表项内容，使其映射到新的物理页面 pp，同时更新 TLB 缓存，并增加物理页

面的引用计数。
函数返回 0，表示成功。
如果页表项不存在或为空，继续执行下一步。
清除 TLB 缓存并重新获取页表项指针：
调用 tlb_invalidate 函数清除 TLB 缓存，以确保新的映射能够立即生效。
再次调用 pgdir_walk 获取虚拟地址 va 对应的页表项指针，这次允许创建新的页表项，将结果保存在 pgtable_entry 中。

检查是否成功获取页表项：

如果 pgtable_entry 为空，表示内存不足，返回错误码 -E_NO_MEM。
设置新的页表项： 将页表项设置为映射到物理页面 pp 的物理地址，并设置相应的权限，同时增加物理页面的引用计数。返回成功状态：返回 0 表示成功执行映射操作。
完成结果如图 9 所示。

```
303 int page_insert(Pde *pgdir, struct Page *pp, u_long va, u_int perm)
304 {
305     u_int PERM;
306     Pte *pgtable_entry;
307     PERM = perm | PTE_V;
308     int ret;
309     /* Step 1: Get corresponding page table entry. */
310     pgdir_walk(pgdir, va, 0, &pgtable_entry);
311
312     if (pgtable_entry != 0 && (*pgtable_entry & PTE_V) != 0) {
313         if (pa2page(*pgtable_entry) != pp) {
314             page_remove(pgdir, va);
315         } else  {
316             tlb_invalidate(pgdir, va);
317             *pgtable_entry = (page2pa(pp) | PERM);
318             return 0;
319         }
320     }
321     tlb_invalidate(pgdir,va);
322     pgdir_walk(pgdir, va, 1, &pgtable_entry);
323     if(pgtable_entry == 0){
324         return -E_NO_MEM;
325     }
326
327     *pgtable_entry = (page2pa(pp)) | PERM;
328     pp->pp_ref++;
329     return 0;
330 }
```

图 9　page_insert 的完成结果

Exercise 2.8

> **Exercise 2.8** 完成 mm/tlb_asm.S 中 tlb_out 函数。 ■

这里我们实现了一个 TLB 失效操作，主要用于在 MIPS 架构中进行 TLB 的操作。以下是对代码的文字解释：
TLB 失效操作：
mfc0 k1, CP0_ENTRYHI: 将 CP0 寄存器 EntryHi 的值读入寄存器 k1。
mtc0 a0, CP0_ENTRYHI: 将参数 a0 的值写入 CP0 寄存器 EntryHi，即设置新的 TLB Entry。
tlbp: 执行 TLB Probe 指令，用于查找 TLB 中是否存在与当前 EntryHi 匹配的条目。
mfc0 k0, CP0_INDEX: 将 TLB Index 读入寄存器 k0。

检查 TLB 查找结果：
bltz k0, NOFOUND: 如果 TLB Index 小于零（负值），即未找到匹配的 TLB 条目，跳转到标签 NOFOUND 处。
处理 TLB 查找结果为找到的情况：
mtc0 zero, CP0_ENTRYHI: 将零值写入 CP0 寄存器 EntryHi。
mtc0 zero, CP0_ENTRYLO0: 将零值写入 CP0 寄存器 EntryLo0。
tlbwi: 执行 TLB Write Index 操作，将新的 TLB Entry 写入 TLB 中。
标签 NOFOUND：
mtc0 k1, CP0_ENTRYHI: 将之前保存的 EntryHi 的值重新写入 CP0 寄存器 EntryHi。
跳转返回：
jra: 跳转到调用该函数的位置，即函数返回。

完成的代码如下：

```
#include <asm/regdef.h>
#include <asm/cp0regdef.h>
#include <asm/asm.h>

LEAF(tlb_out)
//1: j 1b
nop
    mfc0    k1,CP0_ENTRYHI
    mtc0a0,CP0_ENTRYHI
    nop
    // insert tlbp or tlbwi
    tlbp
    nop
    nop
    nop
    nop
    mfc0    k0,CP0_INDEX
    bltz  k0,NOFOUND
    nop
    mtc0zero,CP0_ENTRYHI
    mtc0zero,CP0_ENTRYLO0
    nop
    // insert tlbp or tlbwi
    tlbwi

NOFOUND:

    mtc0k1,CP0_ENTRYHI

    j     ra
    nop
END(tlb_out)
```

Thinking 部分：

Thinking 2.1

> **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?

解答：

使用虚拟地址查询缓存的可能性：

虚拟地址映射关系：在页表的帮助下，虚拟地址可以被映射到物理地址，从而确定对应的缓存块。虚拟地址标签：缓存的每个块通常包含一个标签，标识该块存储的数据在内存中的位置。虚拟地址可以通过标签匹配来确定是否命中缓存。

好处：

地址空间隔离：使用虚拟地址可以隔离不同进程的地址空间，确保缓存的有效性不受到其他进程的影响。更灵活的内存管理：虚拟地址允许更灵活的内存管理，例如使用虚拟内存技术，这可以提供更大的地址空间。

坏处：

TLB Miss：在使用虚拟地址时，可能发生 TLB 缺失，导致需要进行页表查找和地址转换，增加访问时间。

上下文切换：虚拟地址需要在上下文切换时重新映射，可能导致缓存失效，需要重新加载新的数据。

使用物理地址查询缓存的优势：

无需地址转换：物理地址不需要进行地址转换，无需访问页表，避免了 TLB 缺失带来的额外开销。

更快速的访问：物理地址直接映射到硬件缓存，省去了虚拟地址到物理地址的映射步骤，加速了缓存的访问速度。

Thinking 2.2

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) ■

```
LEAF(tlb_out_optimized)
    //1: j 1b
    nop
    mfc0    k1, CP0_ENTRYHI
    mtc0    a0, CP0_ENTRYHI
    nop

    // insert tlbp or tlbwi in parallel
    tlbp
    mtc0    zero, CP0_ENTRYHI
    nop

    // Check TLB Index in parallel
    mfc0    k0, CP0_INDEX
    bltz    k0, NOFOUND
    nop

    mtc0    zero, CP0_ENTRYHI
    mtc0    zero, CP0_ENTRYLO0
    nop

    // insert tlbp or tlbwi in parallel
    tlbwi
    mtc0    k1, CP0_ENTRYHI

NOFOUND:
    j   ra
    nop
END(tlb_out_optimized)
```

Thinking 2.3

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。

```
1 ^^Iint x;
2 ^^Ichar* value = return_a_pointer();
3 ^^I*value = 10;
4 ^^Ix = (int) value;
```

■

解答：**虚拟地址**

Thinking 2.4

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？

■

解答：

使用 do { /* ... */ } while (0) 的形式来定义宏的函数体，是为了解决在使用宏时可能引发的一些问题，主要包括以下几个方面的好处：

避免语法错误：在使用宏时，宏展开后的代码块可能会嵌套在其他代码中。如果宏的定义不使用 do { /* ... */ } while (0) 形式，而是直接写成 { /* ... */ }，在某些情况下可能导致语法错误。使用 do { /* ... */ } while (0) 可以确保宏展开后始终形成一个语句，避免由于宏的使用而导致语法问题。

防止条件编译引发的问题：在条件编译中，如果一个宏展开后的代码块没有被包裹在 do { /* ... */ } while (0) 中，而是直接写成 { /* ... */ }，在条件编译中可能会导致语法错误或者逻辑错误。使用 do { /* ... */ } while (0) 确保条件编译时展开后的代码仍然是一个语句。

增强宏的使用安全性：使用 do { /* ... */ } while (0) 可以避免一些潜在的问题，比如在宏展开后多次使用分号 (;)，而不会引发语法错误。这有助于确保在使用宏时，宏展开的结果始终是一个完整的语句。

避免副作用问题：如果一个宏的定义不使用 do { /* ... */ } while (0)，而是直接写成 { /* ... */ }，在一些使用宏的场景中，可能会引发副作用问题。使用 do { /* ... */ } while (0) 可以确保宏展开后的代码块在逻辑上是一个独立的语句，避免潜在的副作用问题。

Thinking 2.5

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。

■

在 mm/pmap.c 中的实现了与物理内存页的管理，其中包括以下操作：

物理内存页的分配：实现了物理内存页的分配过程。当需要分配一个物理内存页时，系统会从空闲列表中选择一个可用的页，并将其相关信息填充到一个 Page 结构体中，表示该物理内存页的元数据。

物理内存页的释放： 当一个物理内存页不再被使用时，系统会通过某种方式将其释放，可能会更新空闲列表或者其他管理结构。相应的 Page 结构体中的信息也会进行更新，例如将引用计数减少，标记为可用等。

页表的使用： 操作系统使用页表来建立虚拟地址与物理地址之间的映射关系。通过页表，可以根据虚拟地址找到对应的物理内存页。Page 结构体包含了与页表项的关联，以便更快地定位和管理物理内存页。

Thinking 2.6

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)。

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

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

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

梳理：

根据 include/queue.h 和 include/pmap.h 的结构定义，我们可以梳理出 Page 列表的结构：

include/queue.h 中的结构定义：

struct element: 通用链表元素结构，包含一个指向下一个元素的指针 next。

struct queue: 通用队列结构，包含指向队列头和尾的指针 head 和 tail。

include/pmap.h 中的结构定义：

struct Page: 表示物理内存页的结构体，包含了物理内存页的相关信息，并嵌入了一个链表元素 struct element page_list_elem。

struct page_list: 表示 Page 列表的头结构，包含一个队列结构 struct queue pages。

根据梳理结果我们可以知道此题选 C。

Thinking 2.7

> **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？ ■

在 mmu.h 中定义的 bzero 函数中，函数的参数都是虚拟地址，因为在高级编程语言中，通常使用**虚拟地址**来引用内存。

Thinking 2.8

> **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？ ■

虚存起始地址为 0xC0000000，表示最高的 2GB 是内核地址空间。因此，页目录的起始地址可以通过将虚存起始地址与一个页表的大小（4MB）相加得到。因此，Win2k 内核的页目录的起始地址是 0xC0400000。

Thinking 2.9

> **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。 ■

跳转到 NOFOUND 的流程：
首先，通过 tlbp 指令查找 TLB 表，判断虚拟地址是否在 TLB 中有对应的物理地址映射。
如果找到了映射（bltz 指令的结果不小于 0），则跳转到 NOFOUND 标签之后的指令。
如果未找到映射，则执行 NOFOUND 标签之前的指令，清空 TLB 的相关寄存器，然后跳转到 NOFOUND 标签之后的指令。

tlbp 和 tlbwi 指令：
tlbp 指令用于在 TLB 中查找虚拟地址的物理地址映射。
tlbwi 指令用于将一个新的 TLB 表项写入 TLB 中。

为何第 10 行处指令后有 4 条 nop 指令：
在 MIPS 指令集中，一些指令需要在执行后等待一些时钟周期以确保结果正确。
这些 nop 指令可能是为了在流水线上插入等待周期，以确保在执行 tlbp 后的操作时，TLB 的状态已经准备好。
插入 nop 指令是为了调整指令执行的时序，以适应硬件的需要，确保后续指令能够正确执行。

Thinking 2.10

> **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。 ■

***pgtable_entry = (pa + i) | perm | PTE_V;**
其中 pa 是物理地址，而 pgtable_entry 所指向的地址是通过 boot_pgdir_walk 函数计算得到的，而 boot_pgdir_walk 内部使用了 KADDR 宏，将物理地址转换为虚拟地址。因此，pa + i 实际上是对虚拟地址进行的加法操作，而不是对物理地址。

为了修复这个问题，应该将 pa + i 转换为虚拟地址，可以使用 PADDR 宏来实现：
***pgtable_entry = (PADDR(pa + i)) | perm | PTE_V;**
PADDR 宏将虚拟地址转换为物理地址，确保正确的地址计算。这个修改应该能够解决代码中可能导致的问题，确保正确的页表项被写入。

Thinking 2.11

> **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。 ■

在 x86 体系结构下，页表的组织形式在启用 PSE（Page Size Extension）时会发生变化。PSE 位是 CR4 寄存器的一个标志位，当 PSE 位被设置为 1 时，系统开启了 4MB 大页面模式。

普通模式（PSE 未启用）：
页目录项（Page Directory Entry）指向页表，页表项（Page Table Entry）指向 4KB 的物理页面。
使用两级页表结构。

启用 PSE（PSE 位设为 1）：
页目录项指向 4MB 的物理页面，而不是页表。
不再需要页表，因为一个页目录项就可以指向一个 4MB 的物理页面。
使用一级页表结构，减少了一级的间接寻址。

总体而言，启用 PSE 后，可以通过更少的页表项来映射相同的虚拟地址范围，从而提高了页表的查找效率。这主要是通过增加每个页目录项指向的物理页面的大小，减少了间接寻址的层次，提高了访存效率。但是需要注意的是，PSE 模式下的大页面无法进行精细的页面控制，因为一页就映射了 4MB 的地址范围。

本次实验耗时 7 小时