> 往届链接在最后

> - 所有的应用程序看到的都是4GB虚拟空间，内核使用kseg0，用户使用kuseg，但实际上所有的应用无论在哪个seg，都通过虚拟映射到了MOS中仅有的64MB空间
> - MOS中的 `PADDR` 与 `KADDR` 这两个宏可以对位于 kseg0 的虚拟地址和对应的物理地址进行转换。对于位于 **kuseg 的虚拟地址**，**MOS中采用两级页表**结构进行地址转换，即 kuseg 才需要用到页表
> - 对页表进行操作时，硬件处于**内核态**，即页表的映射也是**线性映射**
> - 在MOS中，若**物理页全部被映射**，不会申请新的物理项，而是简化为直接从 `page_alloc()` 中报错 `-E_NO_MEM`
> - 软件操作 TLB 的流程总是分为两步：1. 填写 CP0 寄存器 2. 使用 TLB 相关指令

> 映射与寻址规则（内存布局）
>
> - 若虚拟地址处于 `0x80000000~0x9fffffff` (`kseg0`)，则将虚拟地址的**最高位**置 0 得到物理地址，通过 cache 访存。**这一部分用于存放内核代码与数据**。
> - 若虚拟地址处于 `0xa0000000~0xbfffffff` (`kseg1`)，则将虚拟地址的**最高 3 位**置 0 得到物理地址，不通过 cache 访存。**这一部分可以用于访问外设**。
> - 若虚拟地址处于 `0x00000000~0x7fffffff` (`kuseg`)，则需要**通过 TLB 转换**成物理地址，再通过 cache 访存。**这一部分用于存放用户程序代码与数据**。
>
> 内存管理相关的 CP0 寄存器
>
> | 寄存器序号 | 寄存器名 | 用途 |
> | --- | --- | --- |
> | 8 | `BadVaddr` | 保存引发地址异常的虚拟地址 |
> | 10、2、3 | `EntryHi`、`EntryLo0`、`EntryLo1` | 所有读写 TLB 的操作都要通过这三个寄存器，详见下一小节 |
> | 0 | `Index` | TLB 读写相关需要用到该寄存器 |
> | 1 | `Random` | 随机填写 TLB 表项时需要用到该寄存器 |

物理内存管理流程

```
------- 4GB 的实际物理空间为 64MB
|kseg2|
-------
|kseg1|
-------
|kseg0|
|     |
```

```
|      | 页目录 + 页表
|      | 都申请且映射在此
|      |
~~~~~~~0x80430000
|pages| 页控制块
~~~~~~~0x80400000
|.text|
|.data| 存有 page_free_list，cur_pgdir
|.bss |
-------0x80000000
|      |
| ... |
|kuseg|
| ... |
|      |
-------

PS：页控制块 pages 指向所有的64MB中的所有 Page
PS：page_free_list 为链表结构体类型，以双向链表空闲页管理块
PS：cur_pgdir 在tests测试中出现，从空闲页链表中申请一个地址块存放，即 cur_pgdir 指向页目录基址的虚拟地址
```

- `mips_detect_memory(u_int _memsize)`（`kern/pmap.c`）探测硬件的可用内存，其内存值已经由 `bootloader` 的 `ram_low_size` 返回，直接使用

  进行初始化 `memsize`：总物理内存对应的字节数

  得到 `npage`：总物理页数

- `mips_vm_init()`（`kern/pmap.c`）调用 `alloc()` 函数初始化内存管理数据结构的空间分配（建立页式内存管理机制之前使用，建立数据结构管理内存）

  `void *alloc(u_int n, u_int align, int clear)`：分配 `n` 字节的空间并返回初始的虚拟地址，同时将地址按 `align` 字节对齐，若 `clear` 参数为真，则将申请的内存值清零

- `page_init()`（`kern/pmap.c`）初始化空闲页面链表 `page_free_list`，存储"没有被使用"的页控制块，进行页对齐，然后将已使用的空间对应的所有物理页面的也控制块的引用次数 `pp_ref` 标为 `1`，没有使用的标为 `0`

- `page_alloc(struct Page **new)`（`kern/pmap.c`）用于在MOS中申请存储空间，有空闲页面时取出链表头部的一页，将该页对应的页控制块的地址放到调用者指定的地方

- `page_decref(struct Page *pp)`（`kern/pmap.c`）将 `pp` 对应页控制块的引用次数减 1，如果引用次数变为 0，则调用 `page_free()` 将对应物理页面重新设置为空闲页面

`page_free(struct Page * pp)` 将 `pp` 指向的也控制块重新插入到 `page_free_list` 中

虚拟内存管理流程

> 当 `page_lookup()` 函数在页表中找不到对应表项时，调用 `passive_alloc()`（`kern/tlbex.c`）函数进行处理，若该虚拟地址合法，会为此虚拟地址申请一个物理页面（`page_alloc()`），并将虚拟地址映射到该物理页面（`page_insert`），即进行被动页面分配

- `pgdir_walk(Pde *pgdir, u_long va, int create, Pte **ppte)`（`kern/pmap.c`）将一级页表基地址 `pgdir` 对应的两级页表结构中 `va` 虚拟地址所在的二级页表的指针存储在 `ppte` 指向的空间上（得到的二级页表的物理地址转化为在 `kseg0` 中的虚拟地址）**（顺带会创建二级页表）**
- `page_insert(Pde *pgdir, u_int asid, struct Page *pp, u_long va, u_int perm)`（`kern/pmap.c`）将一级页表基地址 `pgdir` 对应的两级页表结构中虚拟地址 `va` 映射到页控制块 `pp` 对应的物理页面，并将页表项权限设为 `perm`**（会将当前虚拟地址与申请到的真正对应的物理地址建立联系）**【guidebook P77】
- `struct Page *page_lookup(Pde *pgdir, u_long va, Pte **ppte)`（`kern/pmap.c`）返回一级页表基地址 `pgdir` 对应的两级页表结构中虚拟地址 `va` 映射的物理页面的页控制块，同时将 `ppte` 指向的空间设为对应的二级页表项地址
- `void page_remove(Pde *pgdir, u_int asid, u_long va)`（`kern/pmap.c`）删除一级页表基地址 `pgdir` 对应的两级页表结构中虚拟地址 `va` 对物理地址的映射

TLB 维护流程

- **TLB 相关指令**
  - `tlbr`：以 `Index` 寄存器中的值为索引，读出 TLB 中对应的表项到 `EntryHi, EntryLo0, EntryLo1`
  - `tlbwi`：以 `Index` 寄存器中的值为索引，将此时 `EntryHi, EntryLo0, EntryLo1` 的值写到索引指定的 TLB 表项中
  - `tlbwr`：将 `EntryHi, EntryLo0, EntryLo1` 的数据随机写到一个 TLB 表项中（此时 `Random` 寄存器来“随机”指定表项）
  - `tlbp`：根据 `EntryHi` 中的 `Key`（包含 VPN 和 ASID），查找 TLB 中与之对应的表项，并将表项的索引存入 `Index` 寄存器（若未找到匹配项，则 `Index` 最高位置被置 1）
- **TLB维护流程**

- **TLB 旧表项无效化**：更新页表中虚拟地址对应的页表项的同时，将 TLB 中对应的旧表项无效化

  `tlb_invalidate(u_int asid, u_long va)`（`kern/tlbex.c`）删除特定虚拟地址在 TLB 中的旧表项（主要逻辑依靠位于 `kern/tlb_asm.S` 中的 `tlb_out` 函数）

- **TLB 重填**：在下一次访问该虚拟地址时，硬件会触发 TLB 重填异常，此时操作系统对 TLB 进行重填

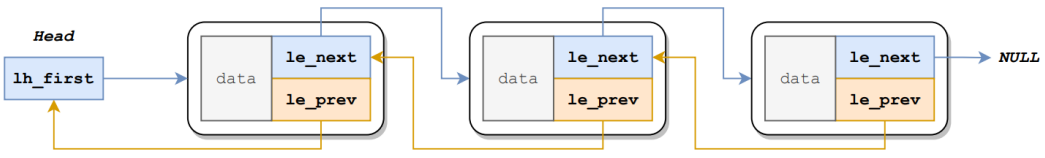
  `_do_tlb_refill(u_long *pentrylo, u_int va, u_int asid)`（`kern/tlbex.c`）

  重填过程主要由 `kern/tlb_asm.S` 中的 `do_tlb_refill` 函数完成，由于奇偶页设计，该过程需要重填触发异常的页面，及其邻居页面。将两个页面对应的页表项先写入 `EntryLo` 寄存器，再填入 TLB

链表宏

- 注意：宏需要注意都写在**同一行**，使用 `\` 进行不同行的组织
- `Page_list` 结构体架构

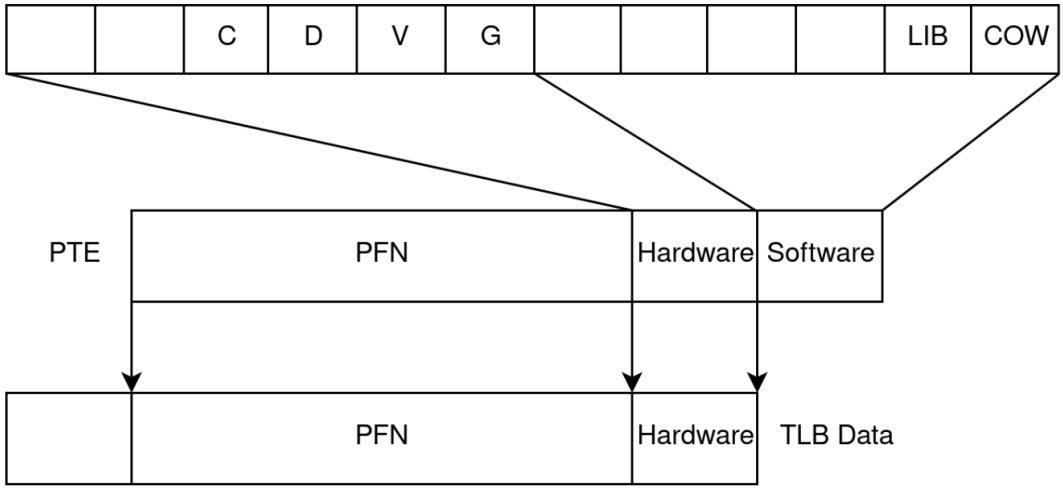
```
struct Page_list{
    struct Page {
        struct List_entry{
            struct Page *le_next;
            struct Page **le_prev;
        } pp_link;
        u_short pp_ref; // 页面引用次数
    }* lh_first; // 指向链表第一个页
}
```



页表及页表项规范

- 页表及页表项规范：
  - 每个页表占用空间为 `4KB`，为一个物理页面大小，故其类型均为 `u_long`，被重命名为 `Pde & Pte`
  - 虚拟地址分配为 10 + 10 + 12
  - 每个页表项 32 位 = 20 位物理页号 + 12 位标志位
  - 标志位 = 高 6 位硬件标志 + 低 6 位软件标志

- 低 6 位不会存入TLB，高 6 位硬件标志位用户存入 `EntryLo` 寄存器，填入TLB时，仅将高 20 位的物理页号和高 6 位的硬件标志存入

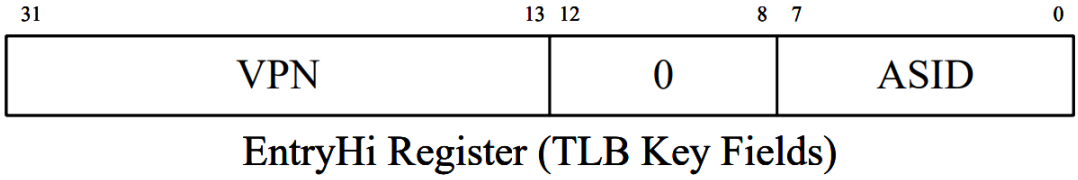


- 注意：使用 `tlb_invalidate` 函数可以实现删除特定虚拟地址的映射，每当页表被修改，就需要调用该函数以保证下次访问相应虚拟地址时一定触发 TLB 重填，进而保证访存的正确性
- 注意：申请新的物理页时，若页面满了，则会使用 FIFO 或者 LRU 等算法进行置换，但MOS中简化了这一过程，会直接在 `page_alloc` 函数中返回 `-E_NO_MEM`

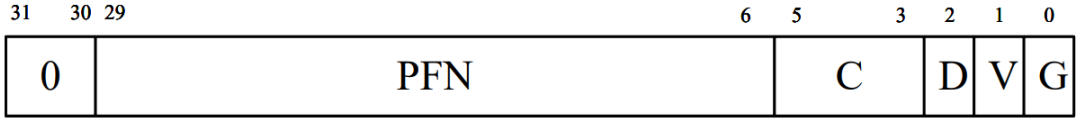
TLB 组成

- `EntryHi, EntryLo0, EntryLo1` 都是 CP0 中的寄存器，只是分别对应 TLB 的 Key 与两组 Data，并不是 TLB 本身

> 4Kc中的TLB采用**奇偶页**设计，VPN 中的高 19 位与 ASID 作为 KEY，一次查找两个 Data（一对相邻页面的两个页表项），并用 VPN 中的最低 1 位在两个 Data 中选择命中的结果
>
> ASID：用于区分不同的地址空间。查找 TLB 表项时，除了需要提供 VPN，还需要提供 ASID（同一虚拟地址在不同的地址空间中通常映射到不同的物理地址）
>
> TLB构建的映射：`< VPN, ASID > -> <PFN, C, D, V, G>`



EntryHi Register (TLB Key Fields)



EntryLo Register (TLB Data Fields)

常用函数及宏总结

include/pmap.c

```
static u_long freemem; 小于它的空间都已分配出去
```

include/mmu.h

```
/*
 * Part 1.  Page table/directory defines.
 */
NASID 256 （可能）表示地址空间标识符（ASID）的数量，用于TLB管理
PAGE_SIZE 4096 页面大小 4KB，2^12B，如0x80001000 =页面=> 0x80001
PTMAP PAGE_SIZE 一个页表项（PTE）映射的物理内存大小 4KB, 2^12B
PDMAP (4 * 1024 * 1024) 一个页目录项（PDE）映射的物理内存大小为 4MB,
2^22B
PGSHIFT 12 页表映射位数，用于确定虚拟/物理地址中页号所需的位移量
PDSHIFT 22 页目录映射位数，用于确定虚拟/物理地址中页目录索引的位移量
PDX(va) ((((u_long)(va)) >> PDSHIFT) & 0x03FF) 页目录项索引编号，如
0x80400000 => 0100000000
PTX(va) ((((u_long)(va)) >> PGSHIFT) & 0x03FF) 页表项索引编号，如
0x80400000 => 1000000000
PTE_ADDR(pte) (((u_long)(pte)) & ~0xFFF) 从页表项或页目录项中提取物理页
基地址，如0x12345678 => 0x12345000
PTE_FLAGS(pte) (((u_long)(pte)) & 0xFFF) 从页表项或页目录项中提取标志位
（低12位），如0x12345678 => 0x00000678

PPN(pa) (((u_long)(pa)) >> PGSHIFT) 获取物理页号
VPN(va) (((u_long)(va)) >> PGSHIFT) 获取虚拟页号

PTE_HARDFLAG_SHIFT 6 PTE/PDE项中Software Flag占位
```

PTE_G (0x0001 << PTE_HARDFLAG_SHIFT) 全局位Global bit。若设置，TLB条目匹配时不检查ASID，即所有进程共享
PTE_V (0x0002 << PTE_HARDFLAG_SHIFT) 有效位Valid bit。若为0，访问该页触发TLB异常
PTE_D (0x0004 << PTE_HARDFLAG_SHIFT) 写使能位Dirty bit。若为0，写操作触发TLB修改异常
PTE_C_CACHEABLE (0x0018 << PTE_HARDFLAG_SHIFT) 控制缓存策略（缓存/非缓存）
PTE_C_UNCACHEABLE (0x0010 << PTE_HARDFLAG_SHIFT) 控制缓存策略（缓存/非缓存）

```
#define KUSEG 0x00000000U
#define KSEG0 0x80000000U
#define KSEG1 0xA0000000U
#define KSEG2 0xC0000000U
```

KERNBASE 0x80020000 内核代码的起始虚拟地址
ULIM 0x80000000 用户空间与内核空间的分界

```
/*
 * Part 3.  Our helper functions.
 */
typedef u_long Pde;
typedef u_long Pte;
```

PADDR(kva) 将内核虚拟地址转化为物理地址，若虚拟地址小于内核与用户空间分界，则panic
KADDR(pa) 将物理地址转化为内核的虚拟地址，若物理页号大于页总量，则panic
assert(x) 断言，否则panic

include/queue.h

```
// 本文件总共包含三种链表操作的宏

/*
 * 双向链表：
 * 链表都是通过一个结构体变量表示，其含有一个元素指向链表的头节点
 * 传入的elm都是指向Page结构体的指针
 */
#define LIST_HEAD(name, type) 创建一个名称为 name 链表的头部结构体，包含一个指向 type 类型结构体的指针，这个指针可以指向链表的首个元素
#define LIST_HEAD_INITIALIZER(head) { NULL }
#define LIST_ENTRY(type) Entry类型定义
        struct {
                struct type *le_next;  /* next element */
```

```
                struct type **le_prev; /* address of previous
next element */
        }
#define LIST_EMPTY(head) ((head)->lh_first == NULL) 判断链表是否为空
#define LIST_FIRST(head) ((head)->lh_first) 返回指向第一个链表内元素的
指针

#define LIST_FOREACH(var, head, field)
        for ((var) = LIST_FIRST((head)); (var); (var) =
LIST_NEXT((var), field))
#define LIST_INIT(head) 初始化链表为空链表，即将链表变量指向的第一个节点置
为NULL
#define LIST_INSERT_AFTER(listelm, elm, field) 将elm元素插入到
listelm元素**后**，field一般为pp_link
#define LIST_INSERT_BEFORE(listelm, elm, field) 将elm元素插入到
listelm元素**前**，field一般为pp_link
#define LIST_INSERT_HEAD(head, elm, field) 插入到链表头部
#define LIST_NEXT(elm, field) ((elm)->field.le_next) 返回指向下一个
链表节点的指针
#define LIST_REMOVE(elm, field) 删除元素
```

include/pmap.h

```
/* 数据及定义 */
extern Pde *cur_pgdir; 全局变量，指向当前进程的页目录首地址

LIST_HEAD(Page_list, Page); 定义链表结构体类型 Page_list，元素指向一个
Page结构体变量地址
typedef LIST_ENTRY(Page) Page_LIST_entry_t; 定义Page的Entry类型为
Page_LIST_entry_t

struct Page {
        Page_LIST_entry_t pp_link;
        u_short pp_ref;
};

extern struct Page *pages; 申请的可用地址空间的首地址
extern struct Page_list page_free_list; 属于Page_LIST_entry_t

/* 函数 */
u_long page2ppn(struct Page *pp); 返回页表项结构体对应的页号
u_long page2pa(struct Page *pp); 返回页表项结构体的物理地址
struct Page *pa2page(u_long pa); 返回物理地址对应的页表项结构体的指针，并
检查是否超出可用页数
u_long page2kva(struct Page *pp); 返回页表项结构体对应的内核虚拟地址
```

`u_long va2pa(Pde *pgdir, u_long va);` 虚拟地址转换为其虚页对应的物理地址（页表首地址）

include/type.h

`#define ROUND(a, n) (((((u_long)(a)) + (n)-1)) & ~((n)-1))` 地址 a，页大小 n，得到向上取整的页初始地址，即下一页的页初始地址
`#define ROUNDDOWN(a, n) (((u_long)(a)) & ~((n)-1))` 地址 a，页大小 n，得到向下取整的页初始地址，即当前页的初始地址

往届上机

注意

- `PTE_V` 无论是在 `Pde` 还是 `Pte` 都需要进行一次判断
- 访问页目录和页表均需要进行一次KADDR的内核地址转换
- **熟练掌握页目录和页表的遍历方式**
- 熟悉 `LIST` 宏的使用，`LIST_EMPTY & LIST_FIRSt & LIST_INSERT_HEAD & LIST_FOREACH` 等
  - `LIST_EMPTY(&page_free_list)`
  - `LIST_FIRST(&page_free_list)`
  - `LIST_INSERT_HEAD(&page_free_list, pp, pp_link)`，其中 `pp` 为 `struct Page *`
  - `LIST_FOREACH(pp, &page_list, pp_link)`，其中 `pp` 为 `struct Page *`

2024-exam

> 给定一个页目录（`pgdir`）以及一个虚拟地址区间 `[va_lower_limit, va_upper_limit)`，请实现一个函数 `page_filter`，要求遍历该地址区间内的所有页表项，统计并返回其中满足以下两个条件的页数：
>
> 1. 页表项必须有效（即具有 `PTE_V` 标志）。
> 2. 对应物理页的引用计数（`pp_ref`）大于或等于一个给定的阈值 `num`。

```
u_int walk_page_list(Pte *pgtable, u_int from, u_int to,
u_int num) {
    u_int res = 0;
    for(int i = from; i < to; i++) {
        Pte * now = (Pte *)(pgtable + i);
        if(!(*now) & PTE_V)) continue;
```

```
            if(pa2page(PTE_ADDR(*now)) -> pp_ref <= num) {
                res++;
            }
        }
        return res;
    }

    u_int page_filter(Pde *pgdir, u_int va_lower_limit, u_int
    va_upper_limit, u_int num) {
        if(va_lower_limit >= va_upper_limit) return 0;
        u_int ld = PDX(va_lower_limit), rd = PDX(va_upper_limit);
        u_int lp = PTX(va_lower_limit), rp = PTX(va_upper_limit);
        u_int ans = 0;
        for(int i = ld; i <= rd; i++) {
            u_int l = (i == ld ? lp, 0), r = (i == rd ? rp,
    1024);
            if(!(pgdir[i] & PTE_V)) continue;
            ans += walk_page_list((Pte
    *)KADDR(PTE_ADDR(pgdir[i]))), l, r, num);
        }
        return ans;
    }
```

2024-extra

实现兄弟系统，分 `4KB` 和 `8KB` 管理内存页块

```
#include <buddy.h>

struct Page_list buddy_free_list[2];
struct Page* buddies[1024];
u_long cnt = 0;
void buddy_init() {
    LIST_INIT(&buddy_free_list[0]);
    LIST_INIT(&buddy_free_list[1]);
    for (int i = BUDDY_PAGE_BASE; i < BUDDY_PAGE_END; i +=
PAGE_SIZE) {
        struct Page *pp = pa2page(i);
        LIST_REMOVE(pp, pp_link);
    }
    for (int i = BUDDY_PAGE_BASE; i < BUDDY_PAGE_END; i += 2 *
PAGE_SIZE) {
        struct Page *pp = pa2page(i);
        LIST_INSERT_HEAD(&buddy_free_list[1], pp, pp_link);
    }
```

```
}

int buddy_alloc(u_int size, struct Page **new) {
    Page_list list;
    u_int mem_size;
    if(size <= PAGE_SIZE) {
        mem_size = 4;
        list = buddy_free_list[0];
    } else {
        mem_size = 8;
        list = buddy_free_list[1];
    }

    if(!LIST_EMPTY(&list)) {
        *new = LIST_FIRST(&list);
        LIST_REMOVE(*new, pp_link);
        return alloc_size >> 2;
    } else if(alloc_size == 4) {
        if(LIST_EMPTY(buddy_free_list + 1)) return -E_NO_MEM;
        else {
            struct Page * head = LIST_FIRST(buddy_free_list + 1);
            LIST_REMOVE(head, pp_link);
            *new = head;
            buddies[cnt++] = head;
            LIST_INSERT_HEAD(&list, pa2page(page2pa(head) +
PAGE_SIZE), pp_link);
        }
    } else return -E_NO_MEM;
}

void buddy_free(struct Page *pp, int npp) {
    if(npp == 1) {
        int flag = 0; // 尝试找到空闲的兄弟页
        for(int i = 0; i < cnt; i++) {
            struct Page * A = NULL;
            int low = -1;
            if(pp == buddy[i]) {
                A = pa2page(page2pa(pp) + PAGE_SIZE);
                low = 1;
            } else if(pa2page(page2pa(pp) - PAGE_SIZE) ==
buddy[i]) {
                A = pa2page(page2pa(pp) - PAGE_SIZE);
                low = 0;
            }
            if(low == -1) continue;
```

```
            struct Page * buddy;
            LIST_FOREACH(buddy, buddy_free_list + 0, pp_link) {
                if(buddy == A) {
                    buddies[i] = NULL;
                    LIST_REMOVE(A, pp_link);
                    LIST_INSERT_HEAD(buddy_free_list + 1, low ?
pp : A, pp_link);
                    flag = 1;
                    break;
                }
            }
            if(flag) break;
        }
        if(!flag) {
            LIST_INSERT_HEAD(buddy_free_list + 0, pp, pp_link);
        }
    } else {
        LIST_INSERT_HEAD(buddy_free_list + 1, pp, pp_link);
    }
}
```

2023-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：
>
> 页表项有效；
> 页表项映射的物理地址为给定的 Page *pp 对应的物理地址；
> 页表项的权限包含给定的权限 perm_mask。

```
u_int page_perm_stat(Pde *pgdir, struct Page *pp, u_int
perm_mask) {
    u_int cnt = 0;
    for(int i = 0; i < 1024; i++) {
        u_int now = pgdir[i];
        if(!(now & PTE_V)) continue;
        Pte * pgtable = KADDR(PTE_ADDR(now));
        for(int j = 0; j < 1024; j++) {
            u_int page = pgtable[j];
            if(!(page & PTE_V)) continue;
            if(page2pa(pp) == PTE_ADDR(page) && ((page &
perm_mask) == perm_mask)) {
                cnt++;
            }
```

```
            }
        }
        return cnt;
    }
```

2023-extra

实现交换系统，内存不够时从特定的物理地址进行换出，之后要使用也可以换入

```
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) */
        struct Page * toSwap = pa2page(SWAP_PAGE_BASE);
        u_int da = disk_alloc();
        for(int i = 0; i < 1024; i++) {
            if(!(pgdir[i] & PTE_V)) continue;
            Pte * pgtable = (Pte *)KADDR(PTE_ADDR(pgdir[i]));
            for(int j = 0; j < 1024; j++) {
                if(!(pgtable[j] & PTE_V)) continue;
                // 不是要交换的页
                if(pa2page(PTE_ADDR(pgtable[j])) != toSwap)
continue;
                pgtable[j] |= PTE_SWP;
                pgtable[j] ^= PTE_V;
                pgtable[j] &= 0xFFF;
                // 这个得看题目要求的硬盘返回值以及硬盘分区方法
                pgtable[j] |= ((da >> 12) << 12);

                tlb_invalidate(asid, i << 22 | j << 12);
            }
        }
        memcpy((void *)da, page2kva(toSwap), PAGE_SIZE);
        LIST_INSERT_HEAD(&page_free_swapable_list, toSwap,
pp_link);
    }

    // Step 2: Get a free page and clear it
    struct Page *pp = LIST_FIRST(&page_free_swapable_list);
    LIST_REMOVE(pp, pp_link);
    memset((void *)page2kva(pp), 0, BY2PG);

    return pp;
}
```

```
// Interfaces for 'Active Swap In'
static int is_swapped(Pde *pgdir, u_long va) {
    /* Your Code Here (2/3) */
    Pde * pgdir_entry = pgdir + PDX(va);
    // va对应的页表需要存在，但被换出
    if(!((*pgdir_entry) & PTE_V)) return 0;
    Pte * pgtable = (Pte *)KADDR(PTE_ADDR(*pgdir_entry));
    Pte * pgtable_entry = pgtable + PTX(va);
    return !((*pgtable_entry) & PTE_V) && ((*pgtable_entry) &
PTE_SWP);
}

static void swap(Pde *pgdir, u_int asid, u_long va) {
    /* Your Code Here (3/3) */
    struct Page * pp = swap_alloc(pgdir, asid);
    Pte * pgtable = (Pte *)KADDR(PTE_ADDR(pgdir[PDX(va)]));
    u_int da = (pgtable[PTX(va)] >> 12) << 12;
    memcpy((void *)page2kva(pp), (void *)da, PAGE_SIZE);
    for(int i = 0; i < 1024; i++) {
        Pde pgdir_entry = pgdir[i];
        if(!(pgdir_entry & PTE_V)) continue;
        Pte * pgtable = (Pte *)(KADDR(PTE_ADDR(pgdir_entry)));
        for(int j = 0; j < 1024; j++) {
            Pte pgtable_entry = pgtable[j];
            if((pgtable_entry & PTE_SWP) && !(pgtable_entry &
PTE_V)
                && ((pgtable_entry >> 12) == (da >> 12))) {
                    pgtable[j] |= PTE_V;
                    pgtable[j] ^= PTE_SWP;
                    pgtable[j] &= 0xFFF;
                    pgtable[j] |= page2pa(pp);
                    tlb_invalidate(asid, i << 22 | j << 12);
                }
        }
    }
    disk_free(da);
}
```

2022-1-exam

知识点：`LIST_FOREACH` 遍历链表，判断/赋值 `Page` 结构体值

2022-1-extra

实现 2^k 的伙伴系统的分配以及释放，需要递归查找和释放恰好符合要求的 buddy 块

2022-2-exam

反向查找，查找某个给定的物理页在哪些虚拟页中出现过，存储并返回数量

2022-2-extra

实现页面迁移，链表分为快速区和慢速区，对于给定的 `pp` 在对应的区域里边查找是否可以移动，找到后进行内容的 copy，再反向找到所有指向这个物理页面的虚拟页面，对页表项进行修改，权限不变

2021-all

- lab2-1 exam：Page 结构体多维护一个 `status`，通过引用次数以及是否分配进行赋值
- lab2-1 extra：位图管理内存。修改 `page_alloc & page_free`，使用位图每次找最小的页面返回
- lab2-2 exam：自映射。64位的三级页表需要 **39** 位虚拟地址进行表示；通过二级页表的起始虚拟地址得到一级页表的起始虚拟地址；得到页目录第 n 项所对应的二级页表起始虚拟地址
- lab2-2 extra：统计所有的物理页被虚拟页映射的次数，存入数组

```
u_int page_test(Pde * pgdir, struct Page * pp) {
    u_int cnt = 0;
    for(int i = 0; i < 1024; i++) {
        u_int now = pgdir[i];
        if(!(now & PTE_V)) continue;
        Pte * pgtable = (Pte *)(KADDR(PTE_ADDR(now)));
        for(int j = 0; j < 1024; j++) {
            u_int page = pgtable[j];
            if(!(page & PTE_V)) continue;
            if(check(pgtable + j)) {
                page_remove(pgdir, asid, i << 22 | j << 12);
                cnt++;
            }
        }
    }
    return cnt;
}

int check(Pte * ppte) {
    return 1;
```

```
}

void page_remove(Pde * pgdir, u_int asid, u_long va) {
    struct Page * pp;
    Pte * ppte;
    pp = page_lookup(pgdir, va, &ppte);
    if(pp == NULL) {
        return;
    }
    page_decref(pp);
    *ppte = 0;
    tlb_invalidate(asid, va);
}
```