# BUAA_OS_Lab2笔记



**MOS 一共 64MB 物理内存，即16个物理页（一页4KB）。**

## 常见概念和宏的理解

**页控制块**（Page Control Block，简称 PCB）是包含与页面管理相关的各种信息的**结构体**。包含页号、脏位、物理地址、虚拟地址等**关于**该页面的完整信息，**不直接参与虚拟地址到物理地址的转换**，更多的是用于操作系统跟踪和管理物理内存中每个页面的状态。

**页表**是虚拟内存到物理内存的**映射**结构，它通过**页表项**提供虚拟地址和物理地址之间的转换。

在 `include/pmap.h`、`include/mmu.h`、`include/types.h` 中：

- `PDX(va)`：Page Directory Index，将**虚拟地址**（`va`）转化为**页目录**中的偏移量。页目录是**多级页表中的一个层级**，通常用于将虚拟地址映射到页表项，查找遍历页表时常用。获取虚拟地址 `va` 的 31-22 位。
- `PTX(va)`：Page Table Index，将**虚拟地址**（`va`）转化为**页表**中的偏移量。查找遍历页表时常用。可以获取虚拟地址 `va` 的 21-12 位。
- `PTE_ADDR(pte)`：Page Table Entry Address，获取**页表项**（`pte`）中的物理地址。每个页表项包含一个物理地址，即对应虚拟页的物理页框的起始地址。
- `PADDR(kva)`：将位于 kseg0 中的虚拟地址转换为物理地址。
- `KADDR(pa)`：将物理地址转换成位于 kseg0 中的虚拟地址（读取 `pte` 后可进行转换）。

- `va2pa(Pde *pgdir, u_long va)`：查页表，虚拟地址 → 物理地址（调试时常用）。
- `pa2page(u_long pa)`：物理地址 → 页控制块（读取 `pte` 后可进行转换）。**返回的是指针！**
- `page2pa(struct Page *pp)`：页控制块 → 物理地址（填充 `pte` 时常用）。
- `page2kva(pp)` 返回这个控制块代表的 **那一页物理内存的内核虚拟地址**。相当于 `KADDR(page2pa(pp))`。
- `ROUND(a, n)` 用于将地址 a 向上 n 字节对齐。

# 内核程序初始化

在 Lab1 基础上，继续完善内核初始化函数`init/init.c`。其中使用了三个函数：`mips_detect_memory(u_int _memsize)`、`mips_vm_init()`、`page_init()`。

## pmap.c/mips_detect_memory(u_int _memsize)

探测硬件可用内存，并对 `npage` 进行初始化。`u_int _memsize` 由 `bootloader` 传入，表示当前可用的**物理内存**大小。该函数存储**物理内存总字节数**和可用的**总物理页数**。可用的总物理页数 `npage = memsize >> PGSHIFT;`。

## pmap.c/mips_vm_init()

对 `pages` 数组进行空间分配。 `pages` 数组的每个元素都是 `Page` 类型：

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

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

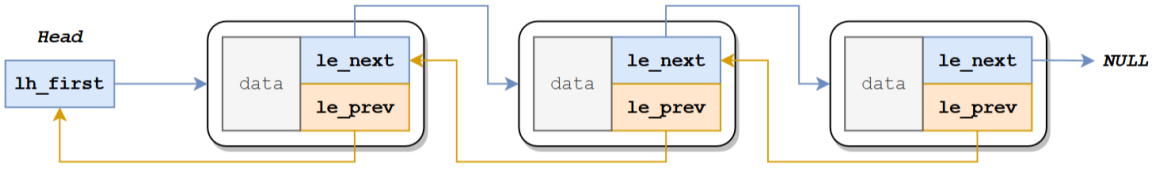
## pmap.c/alloc

`void *alloc(u_int n, u_int align, int clear)`，意思是：返回新分配的长度为n、按 align 对齐的空间的首地址。clear 标志位为1为清空这部分空间。**专用于 `kseg0` 内的虚拟内存分配**，根据 `kseg0` 段的特殊性质，虚拟内存向物理内存线性映射，因此能够间接地实现分配物理内存。

`freemem` 是一个指针，表示“从哪里开始我们还有自由空间可以用”，而这个地址是**虚拟地址**。表示小于 `freemem` 对应物理地址的物理内存都已经被分配了。

## queue.h

**页控制块数组**中的每一项都代表了一页物理内存。需要高效实现**空闲页控制块的申请和释放。**——内核链表（双向链表）。由 `queue.h` 中一系列头文件封装。




所有双向链表操作：

| 宏名 | 作用 | 输入参数说明 | 备注与图示 |
| --- | --- | --- | --- |
| `LIST_HEAD(name, type)` | 定义一个链表头结构体类型 | `name`：链表头名字，我们没用过；`type`：**通常是** `pp_link` | 结果为一个结构体，包含 `type *lh_first` |
| `LIST_HEAD_INITIALIZER(head)` | 初始化链表为空 | `head` **是链表变量名，通常是** `&page_free_list` | 可用于定义时初始化：`= LIST_HEAD_INITIALIZER(mylist)` |
| `LIST_ENTRY(type)` | 在链表节点结构体中定义链表项字段 | `type`：结构体类型 | 会生成 `le_next` 和 `le_prev` 成员，用于双向连接 |
| `LIST_INIT(head)` | 清空链表（将首指针设为 NULL） | `head`：链表头指针（一般 `&page_free_list`） | 常用于初始化链表头：`LIST_INIT(&page_free_list);` |
| `LIST_EMPTY(head)` | 判断链表是否为空 | `head`：链表头指针 | 返回布尔值 |
| `LIST_FIRST(head)` | **获取链表第一个元素** | `head`：链表头指针 | 可用于 `page_alloc()` 取出链头 |
| `LIST_NEXT(elm, field)` | 获取元素 `elm` 的下一个元素 | `elm`：当前元素指针 `field`：链表项字段名（如 `pp_link`） | 其实就是 `(elm)->field.le_next` |
| `LIST_FOREACH(var, head, field)` | 遍历链表元素 | `var`：循环变量 `head`：链表头指针 `field`：链表项字段名 | 用法为 `LIST_FOREACH(p, &page_free_list, pp_link)` |
| `LIST_INSERT_HEAD(head, elm, field)` | 插入元素 `elm` 到链表头部 | `head`：链表头指针 `elm`：要插入的节点 `field`：链表项字段 | 插入后它就是新链表头，旧头移至其后 |
| `LIST_INSERT_AFTER(listelm, elm, field)` | 把元素 `elm` 插入在 `listelm` 之后 | `listelm`：已存在元素 `elm`：待插入元素 `field`：链表项字段 | 要求 `listelm` 已经在链表中 |
| `LIST_INSERT_BEFORE(listelm, elm, field)` | 把元素 `elm` 插入在 `listelm` 之前 | 同上 | 比 `AFTER` 更少用，原理对称 |
| `LIST_REMOVE(elm, field)` | 将元素 `elm` 从链表中移除 | `elm`：要删除的节点 `field`：链表项字段 | 不需要链表头，内部用 `le_prev` 实现高效删除 |

```c
LIST_INIT(&page_free_list);                          // 初始化链表为空
LIST_INSERT_HEAD(&page_free_list, &pages[0], pp_link);  // 插入 pages[0] 到链表头
LIST_REMOVE(&pages[0], pp_link);                     // 从链表中删除 pages[0]
struct Page *p;
LIST_FOREACH(p, &page_free_list, pp_link) {
    // 在这里你可以使用 p 来访问每一个空闲页控制块
    printk("page index: %u, pp_ref = %u\n", page2ppn(p), p->pp_ref);
}
```

## `pmap.c/page_init(void)`

使用内核链表，建立了**管理物理页面**分配的数据结构 `page_free_list` 。`page_init(void)` 用于初始化空闲页面链表 `page_free_list` ，用于存储"没有被使用"的页控制块。

首先明确 `page_free_list` 的类型 `struct Page_list` 的展开结构：

```
struct Page_list {
    struct Page {
        struct LIST_ENTRY {
            struct Page *le_next;   // 下一个节点
            struct Page **le_prev;  // 前一个节点的 next 的地址
        } pp_link;                  // 链表项

        u_short pp_ref; // 引用次数，表示有多少虚拟页映射到该物理页
    } *lh_first; // 指向链表中第一个 Page 结构体的指针
};
```

**注意 `freemem` 是虚拟地址！想要获取该地址之下有多少个物理页，必须要 `int size = PADDR(freemem) / PAGE_SIZE;` ！**

理清楚 `pages` 数组和 `page_free_list` 链表之间的关系！他们的元素似乎都是 `Page` 结构体类型。`pages` 数组存储的是所有页控制块，而 `page_free_list` 链表是将空闲物理页对应的 Page 结构体全部插入一个链表中！

🧠为什么 `LIST_INSERT_HEAD(&page_free_list, &pages[i], pp_link)` 总是插入到头部，而不是末尾？因为插入链表头部比插入尾部 **更快**，而且在这个阶段根本 **不需要保证顺序！**

## `pmap.c/page_alloc(struct Page **new)`

将 `page_free_list` 空闲链表**头部页控制块对应的物理页面分配出去**，将其从空闲链表中移除，并清空此页中的数据。

## `pmap.c/page_free(struct Page *pp)`

`pp` 指向的页控制块引用次数为0时，重新插入到 `page_free_list` 的头部。

# 虚拟内存管理

**ASID，全称：Address Space Identifier（地址空间标识符）。** 它是 MIPS 架构中 TLB 中的一个字段，用于区分**不同进程的虚拟地址空间。**每个进程有一套自己的虚拟地址空间，但 CPU 的 TLB 是共享的，**ASID 让同一个虚拟地址（VA）在不同进程中表示不同含义**！

## `pmap.c/pgdir_walk`

二级页表检索函数。**查找 PTE。**

给定指向一级页表 基地址的指针和要查找的虚拟地址 `va`。返回指向二级页表**项**的指针 `ppte`。

凡是涉及到 "寻找是否存在这一页表项" 或 "**Get the page table entry**" 都应使用这个函数。

检查页面是否无效使用 `(*pte & PTE_V) == 0`（pde 也可以，因为结构相同，意思是也可以 `*pgdir_entryp & PTE_V`）。

### `pmap.c/page_insert`

增加地址映射函数。

把给定物理页 `pp` 映射到给定虚拟地址 `va` 上去，并设置访问权限。

### `pmap.c/page_lookup`

寻找映射的物理地址函数。

从页表中"查找"某个虚拟地址 `va` 对应的 **物理页控制块（struct Page *pp）**，还能顺便把这个 `va` 对应的 **PTE（页表项地址）** 也告诉你（通过 `ppte`）。

| 函数名 | 功能 | 输出内容 |
| --- | --- | --- |
| `pgdir_walk()` | 找到 `va` 对应的二级页表**页表项地址** | `*ppte = PTE 地址` |
| `page_insert()` | 把 `pp` 插入到 `va` 对应位置（建立映射） | 无，返回 0 或错误 |
| `page_lookup()` | 查找 `va` 当前映射的是哪一**页**，看看这地址有没有页，有就告诉你是哪页！ | 返回页控制块 `pp`，可选输出 `ppte` |

## TLB 维护与重填

### `tlbex.c/void tlb_invalidate`

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。刚刚执行了 `page_insert()`，更新了页表内容，但是 TLB 中可能还缓存着旧的映射，因此要 flush TLB。

### `tlb_out`

```
LEAF(tlb_out)                   # 宏：定义叶函数，开始
.set noreorder                  # 关闭指令重排序，保证每条指令按顺序执行（对 CP0 寄存器操作特别重要）

    mfc0    t0, CP0_ENTRYHI # 从协处理器 CP0 的 ENTRYHI 寄存器中取值，保存到 t0 中（备份原始值）
    mtc0    a0, CP0_ENTRYHI # 将 a0 的值写入 CP0_ENTRYHI，用于进行 TLB 读取匹配
    nop                     # 写 CP0 寄存器后需要延迟槽，插入 nop 避免错误执行

    /* Step 1: Use 'tlbp' to probe TLB entry */
    /* Exercise 2.8: Your code here. (1/2) */
    tlbp    # 执行 TLB probe 操作，查找 ENTRYHI 是否命中 TLB 中的某个条目

    nop     # tlbp 为延迟槽指令，需要插入 nop 确保指令执行正确

    /* Step 2: Fetch the probe result from CP0.Index */
    mfc0    t1, CP0_INDEX       # 将 CP0_INDEX 中的值读取到 t1，INDEX 保存的是 probe 的结果索引（或最高位为1表示未命中）

.set reorder                    # 允许指令重排序，从此处开始可以优化指令顺序

```

```
    bltz    t1, NO_SUCH_ENTRY  # 如果 t1 < 0，即 probe 没命中，则跳转到 NO_SUCH_ENTRY 标签（跳过写入）

.set noreorder                 # 再次禁止指令重排序，准备写入 CP0

    mtc0    zero, CP0_ENTRYHI  # 清空 CP0_ENTRYHI（设置为 0）
    mtc0    zero, CP0_ENTRYLO0 # 清空 CP0_ENTRYLO0（设置为 0）
    mtc0    zero, CP0_ENTRYLO1 # 清空 CP0_ENTRYLO1（设置为 0）
    nop                        # 再次使用延迟槽，保证上面的 mtc0 写入完成

    /* Step 3: Use 'tlbwi' to write CP0.EntryHi/Lo into TLB at CP0.Index  */
    /* Exercise 2.8: Your code here. (2/2) */
    tlbwi          # 将当前 CP0_ENTRYHI/ENTRYLO0/ENTRYLO1 写入 TLB 中的指定索引项（由 INDEX 给出）

.set reorder       # 恢复指令重排序

NO_SUCH_ENTRY:     # 标签：表示没有找到匹配的 TLB 项
    mt
```

# BUG 记录

如果调用一个分配内存的函数，该函数分配成功返回0，分配失败返回 -E_NO_MEM（如 `page_alloc`、`pgdir_walk`），题目也说了如果分配失败则返回错误，那么千万不能仅仅写一行调用就完事了！一定要在调用方进行判断：

```
if (pgdir_walk(pgdir, va, 1, &pte) == -E_NO_MEM) {
    return -E_NO_MEM;
}
```

# 内存三界：虚拟地址、物理地址、内核虚拟地址

| 名字 | 含义 | 举例 |
| --- | --- | --- |
| `pp` | 页控制块指针 `struct Page*` | 描述某一页 |
| `ppn` | 页号（Page Number） | 第几页（从 0 开始） |
| `pa` | 物理地址（Physical Address） | 0x00100000 |
| `va` | 虚拟地址（Virtual Address） | 0xC0100000 |
| `kva` | 内核虚拟地址（通常高地址） | 属于内核直接映射空间，比如 `kseg0` |
| `pgdir` | 页目录（页表的页表） | 多级页表顶层，用来查虚拟地址 |

```
        虚拟地址 va
            |
            v
    +---------------+
    | 页表查表（va2pa）|
    +---------------+
```

```
                |
                v
          物理地址 pa
                |
      ------------------
     |                  |
     v                  v
内核虚拟地址 kva      页控制块 page*
  (KADDR(pa))         (pa2page)
```

- `page2ppn(struct Page *pp)`：把某个页控制块指针 `pp`（page pointer） 转换成它在 `pages[]` 中的编号（页号，直接减去首地址指针）
- `page2pa(struct Page *pp)`： 页控制块 → 物理地址（把该页页号乘上页面大小）
- `pa2page(u_long pa)`：物理地址 → 指向该页控制块的指针 `pp`
- `page2kva(struct Page *pp)`：页控制块 → 内核虚拟地址（内核能直接访问的），方法是先取其对应的物理地址，再映射成内核虚拟地址！
- `KADDR(pa)`： 物理地址 → 内核虚拟地址（用于访问物理内存），前提是 `pa` 要在合法物理内存范围！

| 函数名 | 输入 | 输出 | 作用 | 类别 |
| --- | --- | --- | --- | --- |
| `page2ppn(pp)` | 页控制块 | 页号 | 查页编号 | 控制块工具 |
| `page2pa(pp)` | 页控制块 | 物理地址 | 控制块 → 物理地址 | 地址变换 |
| `pa2page(pa)` | 物理地址 | 页控制块 | 反查物理页属于哪个控制块 | 地址变换 |
| `page2kva(pp)` | 页控制块 | 虚拟地址 | 能访问这页内容的虚拟地址 | 映射 |
| `KADDR(pa)` | 物理地址 | 虚拟地址 | 把物理页映射进虚拟空间 | 映射，kseg0专用 |
| `PADDR(kva)` | 虚拟地址 | 物理地址 | 把虚拟地址反推到物理 | 映射，kseg0专用 |
| `va2pa(pgdir, va)` | 页目录 + 虚拟地址 | 物理地址 | 查页表得到映射 | 页表查找 |

| 名称 | 作用 | 适用地址 | 是否查页表 | 示例 |
| --- | --- | --- | --- | --- |
| `PADDR(kva)` | **直接映射**虚拟地址 → 物理地址 | 只用于 **kseg0/kseg1** | ❌ 不查页表 | `0xC0100000` → `0x00100000` |
| `va2pa(pgdir, va)` | 一般虚拟地址 → 物理地址（通过页表） | 用户地址、kseg2、映射区等 | ✅ 查页表 | `0x00400000` → `0x001AB000`（需页表） |

# 补充知识

🧠时刻牢记 `freemem` 是虚拟地址！要获取 `freemem` **以下的**页控制块数，要 `int size = PADDR(freemem) / PAGE_SIZE;`。总页控制块数是 `npage` 。

🧠 `page_init` 是 `freemem` 变量最后一次被使用。在 `mips_vm_init` 函数执行完毕后，`alloc` 函数就不会再被调用了，在此之后的"分配空间"操作通过 `page_alloc` 函数完成。

🧠所有 `queue.h` 中的宏，传入参数全都是指针！例如 `LIST_INIT(&page_free_list);`。

🧠在 MIPS 架构中，虚拟地址空间被划分为几段，最常见的三段：

| 段名 | 虚拟地址范围 | 映射方式 | 缓存? | 用途 |
| --- | --- | --- | --- | --- |
| `kuseg` | `0x00000000` ~ `0x7FFFFFFF` | TLB 映射 | ✅ 缓存 | 用户空间 |
| `kseg0` | `0x80000000` ~ `0x9FFFFFFF` | 直接映射 (cached) | ✅ 缓存 | 内核访问物理内存 |
| `kseg1` | `0xA0000000` ~ `0xBFFFFFFF` | 直接映射 (uncached) | ❌ 不缓存 | 内核访问外设、异常敏感访问 |
| `kseg2/kseg3` | `0xC0000000` ~ `0xFFFFFFFF` | TLB 映射 | ✅ 缓存 | 高端内核空间 |

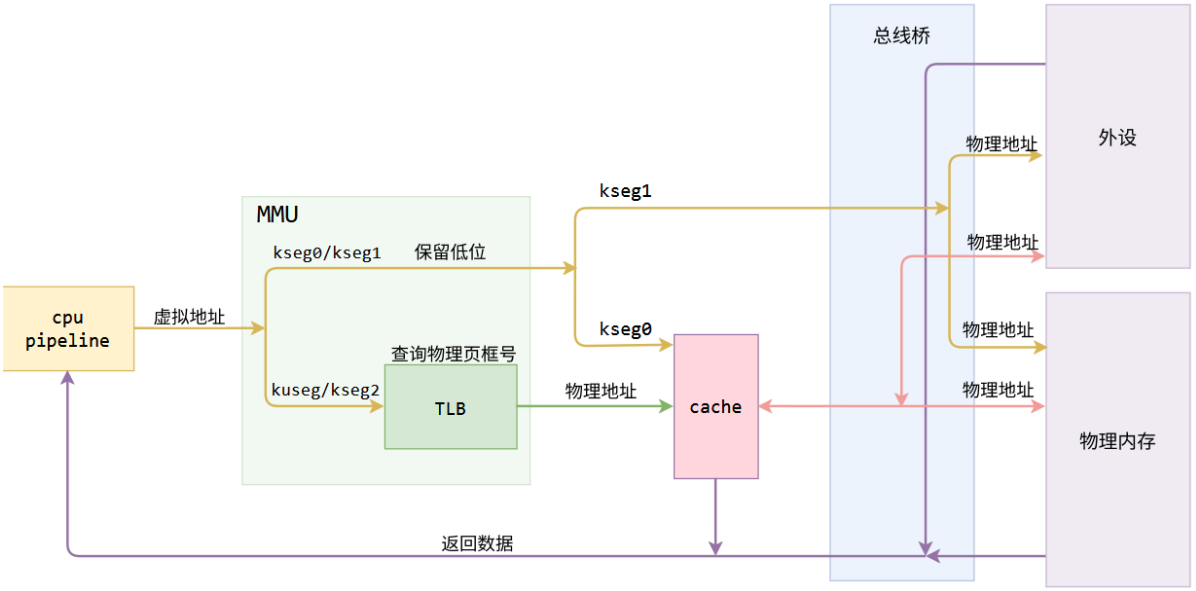



图 2.2: cpu-tlb-memory 关系

# 调试

执行`make test lab=2_1 && make dbg` 进入特定测试点的测试环境；

`layout src` ：按照**源代码**的文本布局对文件进行调试。

`tb mips_init`；

`c`；

从左向右分别为：

1. **继续**：对应 GDB 中的 `continue` 指令
2. **步过**：对应 GDB 中的 `next` 指令
3. **步入**：对应 GDB 中的 `step` 指令
4. **步出**：对应 GDB 中的 `finish` 指令
5. **重启**：对应 GDB 中的 `start` 指令（在程序运行中使用）
6. **停止**：对应 GDB 中的 `kill` 指令

使用 `layout asm` 显示选择 ASM 的 UI 方式（显示**汇编**代码），可以看到运行所在位置；

使用 `break _start` 在程序入口处（_start）位置打一个断点；

使用 `continue`，可以看到正常进入了程序入口。

`si` ： 按汇编指令进行单步调试。

`layout src` ： 按照**源代码**的文本布局对文件进行调试。但是由于 _start 本来就处于汇编代码中，因此在这里执行用源代码布局显示当然不会出现任何东西。因此，回车之后应该在有源代码（C语言）位置处再打一个断点：

`break mips_init`;

然后再 `continue`，就可以进入C语言源代码的 `mips_init` 实现。

# 伙伴系统

```
// Lab 2-1 Extra
struct buddy_allocator {
    int size;
    int used;
    int st_addr;
    int lson, rson;
}buddy[700005];
static int cnt = 1;

#define BLOCKSIZE(i) ((4*(1<<i)) << 10)

void buddy_initialize(int index, int size, int addr) {
    buddy[index].size = size;
    buddy[index].st_addr = addr;
    buddy[index].used = 0;
    buddy[index].lson = 0;
    buddy[index].rson = 0;
}

void buddy_init(void) {
    int i = 0;
    cnt = 1;
    for (i = 0; i < 8; i++) {
        buddy_initialize(cnt++, 10, 32 * (1 << 20) + i * 4 * (1 << 20));
    }
```

```
}

int _buddy_alloc(u_int size, int p, int fa) {
    if (BLOCKSIZE(buddy[p].size) / 2 < size || buddy[p].size == 0) {
 if (!buddy[p].used && BLOCKSIZE(buddy[p].size) >= size && !buddy[p].lson &&
!buddy[p].rson) {
            buddy[p].used = 1;
            return p;
        } else return -1;
    }
    if (buddy[p].used) return -1;
    if (buddy[p].lson == 0 && buddy[p].rson == 0) {
        buddy[p].lson = cnt++;
        buddy[p].rson = cnt++;
        buddy_initialize(buddy[p].lson, buddy[p].size-1, buddy[p].st_addr);
        buddy_initialize(buddy[p].rson, buddy[p].size-1, buddy[p].st_addr +
BLOCKSIZE(buddy[p].size)/2);
    }
    int r;
    r = _buddy_alloc(size, buddy[p].lson, p);
    if (r == -1) r = _buddy_alloc(size, buddy[p].rson, p);
    return r;
}

int buddy_alloc(u_int size, u_int *pa, u_char *pi) {
    int i = 1;
    for (i = 1; i <= 8; i++) {
        int p = i;
        if (buddy[p].used && !buddy[p].lson && !buddy[p].rson) continue;
        int r = _buddy_alloc(size, p, 0);
        if (r != -1) {
            *pa = buddy[r].st_addr;
            *pi = buddy[r].size;
            return 0;
        }
    }
    return -1;
}

void _buddy_free(u_int pa, int i, int fa) {
    if (i == 0) return;
    if (pa == buddy[i].st_addr && !buddy[i].lson && !buddy[i].rson) {
        buddy[i].used = 0;
        return;
    }
    if (pa < buddy[buddy[i].rson].st_addr) _buddy_free(pa, buddy[i].lson,
i);
    else _buddy_free(pa, buddy[i].rson, i);
    if (!buddy[buddy[i].lson].used && !buddy[buddy[i].rson].used &&
            !buddy[buddy[i].lson].lson && !buddy[buddy[i].lson].rson &&
            !buddy[buddy[i].rson].lson && !buddy[buddy[i].rson].rson) {
        buddy[i].used = 0;
        buddy[i].lson = 0;
        buddy[i].rson = 0;
    }
}
```

```
78
79  void buddy_free(u_int pa) {
80      int i;
81      for (i = 1; i <= 8; i++) {
82          if (pa < (i-1) * 4 * (1 << 20) + 32 * (1 << 20)) continue;
83          if (pa >= i * 4 * (1 << 20) + 32 * (1 << 20)) continue;
84          _buddy_free(pa, i, 0);
85      }
86  }
```

# 倒查页表

根据给定的 **物理页** `pp`，在指定的 **页目录** `pgdir` 中查找 **所有映射到该物理页的虚拟页。**最终，把所有被映射到该物理页的**虚拟页号**保存在 `vpn_buffer[]` 数组里，返回匹配的个数。

**如何获取虚拟页号——直接虚拟地址的高20位！**

```
1   int inverted_page_lookup(Pde *pgdir, struct Page *pp, int vpn_buffer[]) {
2       int idx = 0;
3
4       for (int i = 0; i < 1024; i++) {
5           Pde* pgdir_entryp = pgdir + i;
6           if (*pgdir_entryp & PTE_V) {
7               Pte* pgtable = (Pte*)KADDR(PTE_ADDR(*pgdir_entryp));
8
9               for (int j = 0; j < 1024; j++) {
10                  Pte* ppte = pgtable + j;
11                  if (*ppte & PTE_V) {
12                      if (PTE_ADDR(*ppte) == page2pa(pp)) {
13                          vpn_buffer[idx++] = (i << 10) | j;
14                      }
15                  }
16              }
17          }
18      }
19
20      return idx;
21  }
```

# 位图管理法

用一个unsigned int  page_bitmap数组管理内存，要求在该数组中，标号小的元素的低位表示页号小的页面。例如，0号页面由page_bitmap[0]的第0位表示，63号页面由page_bitmap[1]的第31位表示。当只有0号页面与63号页面被占用时，应该有：page_bitmap[0]=0x00000001，page_bitmap[1]=0x80000000。

```
1   unsigned int page_bitmap[(npage + 31) / 32];    // 表示向上取整，也可以直接除32
2
3   void page_init(void) {
4       freemem = ROUND(freemem, PAGE_SIZE);
5
6       /* Mark all memory below freemem as used (set pp_ref to 1) */
7       int size = PADDR(freemem) / PAGE_SIZE;
8       for (int i = 0; i < size; i++) {
```

```
        pages[i].pp_ref = 1;
        page_bitmap[i / 32] |= (1 << (i % 32));  // 将位图中的对应位设置为 1，表示已占用
    }

    /* Mark the remaining pages as free */
    for (int i = size; i < npage; i++) {
        pages[i].pp_ref = 0;
        page_bitmap[i / 32] &= ~(1 << (i % 32));  // 将位图中的对应位设置为 0，表示空闲
    }

    printf("page bitmap size is %d\n", PAGE_BITMAP_SIZE);
}

int page_alloc(struct Page **pp) {
    for (int i = 0; i < PAGE_BITMAP_SIZE; i++) {
        unsigned int mask = page_bitmap[i];
        if (mask != 0xFFFFFFFF) {  // 查找空闲位（位图中为 0 的位置）
            for (int j = 0; j < 32; j++) {
                if ((mask & (1 << j)) == 0) {  // 找到空闲位
                    int page_idx = i * 32 + j;
                    *pp = &pages[page_idx];
                    page_bitmap[i] |= (1 << j);  // 将位图中的对应位设置为 1，表示已分配
                    (*pp)->pp_ref = 1;
                    return 0;
                }
            }
        }
    }
    return -E_NO_MEM;  // 如果没有空闲页
}

void page_free(struct Page *pp) {
    assert(pp->pp_ref == 0);  // 确保引用计数为 0

    int page_idx = page2ppn(pp);  // 获取该页面在数组中的下标

    // 更新位图，标记该页面为空闲
    page_bitmap[page_idx / 32] &= ~(1 << (page_idx % 32));
}
```

# 三级页表

64位操作系统采用三级页表进行虚拟内存管理，每个页表大小为4KB，页表项需要字对齐，其 余条件与二级页表管理32位操作系统相同。请问64位中最少用多少位表示虚拟地址。——39.

输入二级页表的起始虚拟地址`va`，返回一级页表的起始虚拟地址。

```
u_long va_to_pgdir(u_long va) {
    return (va >> 30) << 30;
}
```

输入页目录的虚拟地址`va`和一个整数n，返回页目录第n项所对应的二级页表的起始虚拟地址。

```c
u_long get_second_level_page_table(u_long va, int n) {
    return va + n << 30;
}
```

给定一个一级页表的指针 `pgdir` 和二级页表起始虚拟地址 `va`，`va`为内核态虚拟地址。把合适的地址填写到 `pgdir` 的指定位置，使得 `pgdir` 能够完成正确的自映射。（即计算出`va` 对应的物理地址所在一级页表项位置，并在那里填入正确的页号和权限位）。

```c
u_long cal_page(int func, u_long va, int n, Pde *pgdir) {
    if (func == 3) {  // 任务3：自映射
        int idx = (va >> 30);  // 计算一级页表索引
        Pde *pgdir_entry = &pgdir[idx];  // 获取对应的页目录项
        u_long second_level_addr = va;// va 本身就是二级页表的虚拟地址
        *pgdir_entry = (Pte)second_level_addr | PTE_V | PTE_C_CACHEABLE;
        return 0;
    }
    return 0;
}
```

# 统计物理页面

和倒查页表很像。

给定一个页目录的起始地址，统计在相应的页表中使用物理页面的情况，其中需要对传入的 `cnt` 数组进行修改，使 `cnt[i]` 表示第 `i` 号物理页被页目录下的虚拟页映射的**总次数。**

函数输入的Pde指针的值为页目录的内核虚拟地址，`cnt` 为数组首地址，函数的返回值为 `cnt` 数组的元素个 数，即**物理页的数量（在我们的操作系统中，这个的值为一个常量）**，`cnt[i]` 表示页目录下有 `cnt[i]` 个虚拟页映射到了第i号物理页。

说明：以下代码也展现了给定虚拟地址，如何获取物理页号。要先化为物理地址，再转化为页控制块，再获取物理页号。

```c
int count_page(Pde *pgdir, int *cnt) {
    // 不需要统计页目录本身所使用的物理页，因为页表自映射
    //u_long pd_pa = PADDR(pgdir);              // 页目录的物理地址
    //struct Page *pd_page = pa2page(pd_pa);    // 获取对应页控制块
    //int pd_idx = page2ppn(pd_page);           // 获取页号
    //cnt[pd_idx]++;                            // 页目录本身被用一次

    // 遍历所有页目录项
    for (int i = 0; i < 1024; i++) {
        Pde* pgdir_entryp = pgdir + i;

        if (*pgdir_entryp & PTE_V) {
        cnt[page2ppn(pa2page(*pgdir_entryp))]++;

            for (int j = 0; j < 1024; j++) {
                Pte * ppte = (Pte *)KADDR(PTE_ADDR(*pgdir_entryp)) + j;

                if (*ppte & PTE_V) {
                    cnt[page2ppn(pa2page(*ppte))]++;
                }
            }
```

```
        }
    }

    return npage;  // 返回统计的数组元素个数，即物理页的总数量
}
```