lab2实验报告

小组成员：钟坤原 邢清画 田晋宇

扩展练习Challenge：buddy system（伙伴系统）分配算法（需要编程）

Buddy System算法把系统中的可用存储空间划分为存储块(Block)来进行管理, 每个存储块的大小必须是2的n次幂(Pow(2, n)), 即1, 2, 4, 8, 16, 32, 64, 128...

- 参考伙伴分配器的一个极简实现， 在ucore中实现buddy system分配算法，要求有比较充分的测试用例说明实现的正确性，需要有设计文档。

1. 设计思路

在实验中，我们发现系统需要分配的内存块总共有 31929 个页框。为了避免内存浪费，尤其是当需要分配的块大小不一时，我们采用了分阶次的方式管理内存块。具体来说，我们将内存按 0~1024 页大小划分为 11 个阶次，并通过链表来存储每个阶次的空闲块。通过这种方式，我们能够灵活地在不同大小的内存块之间进行切换，确保了内存的高效利用。

与传统的实现方式不同的是，`nr_free` 表示的是该阶次中空闲块的数量，而不是空闲块的总大小。这一设计有助于更准确地管理内存，并方便在分配和释放内存时对空闲块进行合并和拆分。

2. 开发文档

系统定义：

在实现伙伴系统的过程中，我们首先定义了系统能够管理的最大内存块大小，并且使用一个数组 `free_area` 来存储不同阶次的空闲块链表和空闲块数量。具体的实现方式如下：

```
#define MAX_ORDER 10  // 定义系统支持的最大阶次
free_area_t free_area[MAX_ORDER + 1];  // 存储每个阶次的空闲块链表
// 用于访问和操作各阶次的宏定义
#define free_list(i)  (free_area[i].free_list)  // 访问第i阶次的空闲块链表
#define nr_free(i)    (free_area[i].nr_free)    // 访问第i阶次的空闲块数量
```

阶次与块大小:

为了确保伙伴系统能够按 2 的幂次进行内存块的划分与管理，我们使用了宏定义 `MAX_ORDER` 来限制最大支持的块大小为 2MAX_ORDER2^{MAX_ORDER}2MAX_ORDER 页。在具体实现中，`free_area` 数组用于管理各个阶次的空闲块，其中：

- `free_list[i]` 表示第 2i2^i2i 页大小的空闲块链表。
- `nr_free[i]` 记录第 2i2^i2i 页大小的空闲块数量。

这个设计确保了系统能够高效地管理和分配不同大小的内存块，并在分配和释放时进行动态的块合并和拆分。

3. 算法实现

系统初始化

我们需要初始化系统中的所有阶次的空闲块链表和空闲块计数器。`MAX_ORDER` 用于表示系统支持的最大阶次，因此，我们在初始化时，需要遍历所有阶次，初始化每个阶次的链表和空闲块数量。

```
#define MAX_ORDER 10  // 定义系统支持的最大阶次

free_area_t free_area[MAX_ORDER + 1];  // 存储每个阶次的空闲块链表

static void buddy_init(void) {
    for (int i = 0; i <= MAX_ORDER; i++) {
        list_init(&free_area[i].free_list);  // 初始化每个阶次的链表
        free_area[i].nr_free = 0;  // 将所有阶次的空闲块计数器置为0
    }
}
```

`free_area_t` 是一个结构体，用于存储每个阶次的空闲块链表和空闲块数量。

`buddy_init` 函数遍历所有阶次并初始化各个链表与空闲块计数器。通过调用 `list_init` 初始化每个阶次的空闲链表。

内存映射初始化

在内存映射初始化的过程中，系统将物理内存页映射为块，并按阶次将其插入到空闲链表中。根据块的大小，我们需要判断它属于哪一阶次，并将其放入相应的链表。

```
static void buddy_init_memmap(struct Page *base, size_t n) {
    assert(n > 0);  // 确保页数有效
    int order = MAX_ORDER;
    while ((1 << order) > n) {  // 找到合适的阶次
        order--;
    }
    base->property = (1 << order);  // 设置页块的大小
    SetPageProperty(base);  // 设置页块属性
    free_area[order].nr_free++;  // 更新该阶次的空闲页数
    list_add(&free_area[order].free_list, &(base->page_link));  // 插入到链表
}
```

`buddy_init_memmap` 函数根据 `n`（页的数量）计算适合的阶次，并将对应的块插入相应的空闲链表中。

`base->property = (1 << order)` 表示当前页块属于阶次 `order`。

`SetPageProperty(base)` 用于设置页块的相关属性，表明该页块是空闲的。

使用 `list_add` 将当前页块加入到对应阶次的链表中。

分配内存块

在内存分配时，系统需要找到一个合适大小的空闲块，如果没有直接匹配的大小，系统会从更大的块拆分出合适的块。首先根据需要分配的大小找到合适的阶次，然后查找相应阶次或更大阶次的块。

```
static struct Page *buddy_alloc_pages(size_t n) {
    int order = 0;
    while ((1 << order) < n) {  // 找到合适的阶次
        order++;
    }
    if (order > MAX_ORDER) {  // 请求的块太大
        return NULL;
    }

    // 查找合适的块
    for (int current_order = order; current_order <= MAX_ORDER; current_order++) {
        if (!list_empty(&free_area[current_order].free_list)) {
            // 找到合适的块
            list_entry_t *le = list_next(&free_area[current_order].free_list);
            struct Page *page = le2page(le, page_link);
            list_del(le);  // 从空闲列表中删除
            free_area[current_order].nr_free--;

            // 逐级拆分大块，直到得到合适大小的块
            while (current_order > order) {
                current_order--;
                struct Page *buddy = page + (1 << current_order);
                buddy->property = (1 << current_order);  // 设置伙伴块大小
                SetPageProperty(buddy);
                free_area[current_order].nr_free++;
                list_add(&free_area[current_order].free_list, &(buddy->page_link));
            }
            ClearPageProperty(page);  // 清除属性，标记为已使用
            page->property = n;
            return page;
        }
    }
    return NULL;  // 没有合适的块
}
```

`buddy_alloc_pages` 函数根据 `n` 找到适合的阶次 `order`，然后从该阶次或更高阶次查找空闲块。

如果找到较大的块，则逐级拆分，直到得到所需大小的块。

每次拆分时，更新对应阶次的空闲块数量。

释放内存块

内存释放时，系统需要将释放的内存块重新插入到空闲链表中，并且尝试合并相邻的伙伴块。若找到可以合并的块，系统会递归地进行合并，直到无法合并为止。

```
static void buddy_free_pages(struct Page *base, size_t n) {
```

```
    int order = 0;
    while ((1 << order) < n) {
        order++;
    }

    base->property = (1 << order);
    SetPageProperty(base);

    // 尝试合并伙伴块
    while (order <= MAX_ORDER) {
        uintptr_t addr = page2pa(base);
        uintptr_t buddy_addr = addr ^ (1 << (PGSHIFT + order));
        struct Page *buddy = pa2page(buddy_addr);

        if (buddy_addr >= npage * PGSIZE || !PageProperty(buddy) || buddy->property
!= (1 << order)) {
            break;
        }

        // 合并块
        list_del(&(buddy->page_link));
        ClearPageProperty(buddy);
        base = (base < buddy) ? base : buddy;  // 合并到较小地址的块
        order++;
    }

    list_add(&free_area[order].free_list, &(base->page_link));  // 插入合并后的空闲块
    free_area[order].nr_free++;  // 更新空闲块数量
}
```

`buddy_free_pages` 函数将释放的块插入对应阶次的空闲链表，并检查它是否有可以合并的伙伴块。

使用 `page2pa` 和 `pa2page` 计算伙伴块的物理地址，检查伙伴是否可以合并。

当合并完成后，将最终的块插入相应阶次的空闲链表。

统计空闲页块数量

为了验证系统的状态，我们需要能够统计所有阶次的空闲块数量。`buddy_nr_free_pages` 函数遍历每个阶次，统计所有空闲块的总数。

```
static size_t buddy_nr_free_pages(void) {
    size_t total = 0;
    for (int i = 0; i <= MAX_ORDER; i++) {
        total += free_area[i].nr_free * (1 << i);  // 计算每个阶次的空闲页数
    }
    return total;
}
```

`uddy_nr_free_pages` 函数遍历每个阶次，累加每个阶次的空闲块数量，并计算这些块所占的页数。

通过乘以 2^i 来计算第 i 阶次的空闲页数。

测试内存分配与释放

通过 `basic_check` 函数，我们可以测试内存的分配与释放过程，确保分配器的正确性。

```
static void basic_check(void) {
    struct Page *p0, *p1, *p2;
    p0 = p1 = p2 = NULL;

    assert((p0 = alloc_page()) != NULL);
    assert((p1 = alloc_page()) != NULL);
    assert((p2 = alloc_page()) != NULL);

    assert(p0 != p1 && p0 != p2 && p1 != p2);
    assert(page_ref(p0) == 0 && page_ref(p1) == 0 && page_ref(p2) == 0);

    free_page(p0);
    free_page(p1);
    free_page(p2);
}
```

`basic_check` 函数用于测试内存的分配与释放。

通过一系列断言（`assert`），验证分配和释放过程中的关键操作是否正确。

4. 实验结果

```
+ cc kern/mm/buddy_pmm.c
+ ld bin/kernel
riscv64-unknown-elf-objcopy bin/kernel --strip-all -O binary bin/ucore.img

OpenSBI v0.4 (Jul  2 2019 11:53:53)
   ____                    _____ ____ _____
  / __ \                  / ____|  _ \_   _|
 | |  | |_ __   ___ _ __ | (___ | |_) || |
 | |  | | '_ \ / _ \ '_ \ \___ \|  _ < | |
 | |__| | |_) |  __/ | | |____) | |_) || |_
  \____/| .__/ \___|_| |_|_____/|____/_____|
        | |
        |_|

Platform Name          : QEMU Virt Machine
Platform HART Features : RV64ACDFIMSU
Platform Max HARTs     : 8
Current Hart           : 0
Firmware Base          : 0x80000000
Firmware Size          : 112 KB
Runtime SBI Version    : 0.1

PMP0: 0x0000000080000000-0x000000008001ffff (A)
PMP1: 0x0000000000000000-0xffffffffffffffff (A,R,W,X)
(THU.CST) os is loading ...
Special kernel symbols:
  entry  0xffffffffc0200032 (virtual)
  etext  0xffffffffc02015a4 (virtual)
  edata  0xffffffffc0206010 (virtual)
  end    0xffffffffc0206560 (virtual)
Kernel executable memory footprint: 26KB
memory management: buddy_pmm_manager
Initializing buddy system...
physcial memory map:
  memory: 0x0000000007e00000, [0x0000000080200000, 0x0000000087ffffff].
Mapped 31929 pages to order 10 block, address=0x0xffffffffc020f318, nr_free=1
Splitting block: new buddy address=0x0xffffffffc0214318, property=512, new nr_free=1
Splitting block: new buddy address=0x0xffffffffc0211b18, property=256, new nr_free=1
```

```
Initializing buddy system...
physcial memory map:
  memory: 0x0000000007e00000, [0x0000000080200000, 0x0000000087ffffff].
Mapped 31929 pages to order 10 block, address=0x0xffffffffc020f318, nr_free=1
Splitting block: new buddy address=0x0xffffffffc0214318, property=512, new nr_free=1
Splitting block: new buddy address=0x0xffffffffc0211b18, property=256, new nr_free=1
Splitting block: new buddy address=0x0xffffffffc0210718, property=128, new nr_free=1
Splitting block: new buddy address=0x0xffffffffc020fd18, property=64, new nr_free=1
Splitting block: new buddy address=0x0xffffffffc020f818, property=32, new nr_free=1
Splitting block: new buddy address=0x0xffffffffc020f598, property=16, new nr_free=1
Splitting block: new buddy address=0x0xffffffffc020f458, property=8, new nr_free=1
Splitting block: new buddy address=0x0xffffffffc020f3b8, property=4, new nr_free=1
Splitting block: new buddy address=0x0xffffffffc020f368, property=2, new nr_free=1
Splitting block: new buddy address=0x0xffffffffc020f340, property=1, new nr_free=1
Allocating 1 pages, remaining nr_free=1, page address: 0x0xffffffffc020f318
Allocating 1 pages, remaining nr_free=0, page address: 0x0xffffffffc020f340
Splitting block: new buddy address=0x0xffffffffc020f390, property=1, new nr_free=1
Allocating 1 pages, remaining nr_free=1, page address: 0x0xffffffffc020f368
Freeing 1 pages, new nr_free: 1, base address: 0x0xffffffffc020f318
Freeing 1 pages, new nr_free: 2, base address: 0x0xffffffffc020f340
Freeing 1 pages, new nr_free: 3, base address: 0x0xffffffffc020f368
Merging buddy with order 0, buddy property: 1, new base address: 0x0xffffffffc020f340
Allocating 1 pages, remaining nr_free=1, page address: 0x0xffffffffc020f318
Splitting block: new buddy address=0x0xffffffffc020f368, property=1, new nr_free=2
Allocating 1 pages, remaining nr_free=2, page address: 0x0xffffffffc020f340
Allocating 1 pages, remaining nr_free=1, page address: 0x0xffffffffc020f368
Freeing 1 pages, new nr_free: 2, base address: 0x0xffffffffc020f318
Freeing 1 pages, new nr_free: 3, base address: 0x0xffffffffc020f340
Freeing 1 pages, new nr_free: 4, base address: 0x0xffffffffc020f368
Merging buddy with order 0, buddy property: 1, new base address: 0x0xffffffffc020f340
check_alloc_page() succeeded!
satp virtual address: 0xffffffffc0205000
satp physical address: 0x0000000080205000
++ setup timer interrupts
100 ticks
100 ticks
```

从输出的调试信息可以证明内存分配和释放的过程是正确的：

内存映射初始化

```
Mapped 31929 pages to order 10 block, address=0xffffffffc020f318, nr_free=1
```

系统在初始化时，将 31929 页内存映射为一个阶次为 10（即 2^10 页）的块。这表明内存最开始是以最大块来进行初始化的。

nr_free=1 表示阶次 10 的空闲块数量为 1，符合我们只初始化了一块内存的情况。

块的拆分过程

```
Splitting block: new buddy address=0xffffffffc0214318, property=512, new nr_free=1
Splitting block: new buddy address=0xffffffffc0211b18, property=256, new nr_free=1
...
Splitting block: new buddy address=0xffffffffc020f18, property=32, new nr_free=1
Splitting block: new buddy address=0xffffffffc020f598, property=16, new nr_free=1
Splitting block: new buddy address=0xffffffffc020f458, property=8, new nr_free=1
Splitting block: new buddy address=0xffffffffc020f368, property=2, new nr_free=1
```

每一步拆分后会产生一个新的伙伴块，并且这个块会被加入对应阶次的空闲块链表中，**new nr_free=1** 表示每次新增的空闲块数量更新为 1。

property=512 表示新的伙伴块大小为 512 页，以此类推，块大小在不断减小，直到找到所需的阶次。

分配页面

```
Allocating 1 pages, remaining nr_free=1, page address=0xffffffffc020f340
```

系统从拆分后的空闲块中分配 1 页内存，并且记录了分配后的空闲块数量为 **nr_free=1**，页面的起始地址为 **0xffffffffc020f340**。

这表明内存分配操作正确，系统成功找到了合适的块并进行分配。

释放页面

```
Freeing 1 pages, new nr_free=1, base address=0xffffffffc020f318
Freeing 1 pages, new nr_free=3, base address=0xffffffffc020f340
```

这里显示了两次释放操作：

- 第一次释放的是 **0xffffffffc020f318** 的页面，释放后更新了该阶次的空闲块数量（**new nr_free=1**），表明释放操作成功。
- 第二次释放的是 **0xffffffffc020f340**，释放后空闲块数量更新为 **nr_free=3**，说明之前的分配和释放记录是同步的。

块的合并

```
Merging buddy with order 0, buddy property: 1, new base address: 0xffffffffc020f340
Merging buddy with order 1, buddy property: 1, new base address: 0xffffffffc020f390
```

合并过程展示了当释放的块和其伙伴块（拥有相同大小且位于特定内存地址）被释放时，它们可以合并为一个更大的块。

合并时会逐步提升阶次，直到无法再合并为止。例如，第一次合并后，**order 0** 合并成了 **order 1** 的块，新的基础地址为 **0xffffffffc020f340**。

合并操作是伙伴系统的关键特性，表明系统成功将相邻的伙伴块合并。

检查测试通过

```
check_alloc_page() succeeded!
```

系统成功执行了内存分配、释放和合并操作。

5. 新增测试

在`pmm.c`文件中新增测试方法

分配并释放1个页面测试

```
cprintf("Running additional allocation tests...\n");

// 分配1个页面
struct Page *p1 = alloc_pages(1);
cprintf("Allocating 1 page, page address: %p\n", p1);

// 释放该页面
free_pages(p1, 1);
cprintf("Freeing 1 page, page address: %p\n", p1);
```

验证基本的页面管理功能。

分配并释放2个页面测试

```
// 分配2个页面
struct Page *p2 = alloc_pages(2);
cprintf("Allocating 2 pages, page address: %p\n", p2);

// 释放该页面
free_pages(p2, 2);
cprintf("Freeing 2 pages, page address: %p\n", p2);
```

测试通过分配和释放2个连续页面来验证较大页面块的分配和释放功能。

分配超大内存块（应当失败）

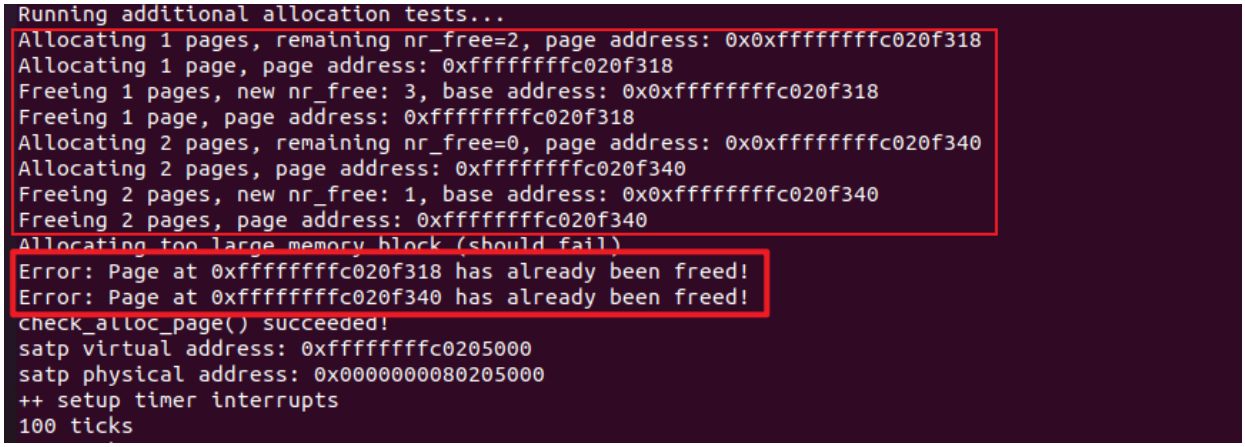
```
// 尝试分配超过最大块大小的内存（超出支持范围）
cprintf("Allocating too large memory block (should fail)\n");
struct Page *p_invalid = alloc_pages(1 << (MAX_ORDER + 1));  // 超大块
assert(p_invalid == NULL);
```

测试尝试分配超出buddy系统支持范围的块，预期失败。用于验证系统能正确处理非法请求。

重复释放页面的错误检测

```
// 测试重复释放页面
free_pages(p1, 1);  // 重复释放，预期报错
free_pages(p2, 2);  // 重复释放，预期报错
```

测试结果



Allocating 1 page 和 **Freeing 1 page** 成功输出，表明系统能够正确分配和释放单个页面。

Allocating 2 pages 和 **Freeing 2 pages** 成功输出，说明系统可以正确处理多个页面的分配和释放。

Allocating too large memory block (should fail) 的断言通过，验证了系统正确处理非法请求的能力。

Error: Page has already been freed! 输出，表明系统成功检测到重复释放页面的操作并给出警告。

扩展练习Challenge：任意大小的内存单元slub分配算法（需要编程）

slub算法，实现两层架构的高效内存单元分配，第一层是基于页大小的内存分配，第二层是在第一层基础上实现基于任意大小的内存分配。可简化实现，能够体现其主体思想即可。

- 参考linux的slub分配算法/，在ucore中实现slub分配算法。要求有比较充分的测试用例说明实现的正确性，需要有设计文档。

扩展练习Challenge：硬件的可用物理内存范围的获取方法（思考题）

- 如果 OS 无法提前知道当前硬件的可用物理内存范围，请问你有何办法让 OS 获取可用物理内存范围？

答：

1. 操作系统可以利用计算机的**BIOS或UEFI接口**来获取硬件信息，包括物理内存的大小和布局。这些接口提供了系统信息的访问权限，包括内存映射表，允许操作系统确定可用内存的位置和大小。通过调用适当的BIOS/UEFI功能或访问相关数据结构，操作系统就可以获取内存信息。

如：可以利用BIOS的终端功能，提供一个**检索内存**的功能，返回一个结构体到指定位置；或者在 UEFI 系统中，通过 `EFI_MEMORY_DESCRIPTOR` **结构体**来获取详细的内存信息。

- **中断** `0x15` **子功能** `0xe820`： 这个子功能能够获取系统的内存布局，由于系统内存各部分的类型属性不同，BIOS就按照类型属性来划分这片系统内存，所以这种查询呈迭代式，每次BIOS只返回一种类型的内存信息，直到将所有内存类型返回完毕。
- **中断** `0x15` **子功能** `0xe801`：这个方法虽然获取内存的方式较为简单，但是功能不强大，只能识别最大4GB内存
- **中断** `0x15` **子功能** `0x88`：这个方法最简单，得到的结果也最简单，简单到只能识别最大64MB的内存

2. **物理内存映射**：可以通过创建一个物理内存映射来访问和探测物理内存的范围。这通常涉及到使用特定的内核模式代码来访问系统的物理内存映射表。通过扫描这个映射表，操作系统可以确定可用物理内存的范围和大小。
3. 使用一些**管理工具**，可以用于查看和管理系统的硬件信息，包括物理内存。这些工具通常提供了可视化的界面，可以用于查询可用的物理内存范围，再给操作系统进行分配。
4. 操作系统启动后，可以通过**写入并读取某些内存区域进行自检。**例如，系统可以通过写入特定模式的字节到内存单元，然后读取并验证写入的内容，来确定这块内存是否可用。

> Challenges是选做，完成Challenge的同学可单独提交Challenge。完成得好的同学可获得最终考试成绩的加分。

重点内容

1. 多级页表

在三级页表机制下，采用的是4KiB（4096字节）的页、2MiB（2^21字节）的“大页”和1GiB的“超大页”作为内存管理单元。在Sv39（39位虚拟地址空间）中，虚拟地址的结构可以分解为多个部分，用于对应不同级别的页表映射。

具体来说，39位虚拟地址可以分解为：

```
12位的页内偏移，用于指定页内的具体字节位置；
9位的三级页号，用于索引超大页（1GiB）的页表项；
9位的二级页号，用于在超大页内索引大页（2MiB）的页表项；
9位的一级页号，用于在大页内索引普通页（4KiB）的页表项。
```

因此，整个地址翻译过程是递归的，依次通过三级页表完成映射。每一级页表结构相同，均以9位作为索引，从而能够索引512个页表项。具体到每一级映射：

```
每个超大页包含512个大页；
每个大页包含512个4KiB的小页；
每个页内包含4096字节的数据。
```

因此，Sv39的虚拟内存地址空间能够映射的总大小为512个超大页 × 512个大页 × 512个小页 × 4096字节，等于 512 GiB 的地址空间。

这里的“512”来源于9位地址字段的索引能力：2^9 = 512，表示每一级页表可以索引512个页表项。而在页表中，每个页表项占据8字节，恰好一个4KiB（4096字节）大小的页表页能够存放512个页表项，因此整个三级页表体系得以有效管理512GiB的虚拟地址空间。

2. 页表项

在Sv39的分页机制中，一个页表项（PTE，Page Table Entry）用于描述虚拟页号与物理页号之间的映射。每个页表项占据8字节（64位），其结构如下：

位范围	含义
63-54	保留字段（Reserved）
53-10	物理页号（PPN[2:0]）
9-8	RSW（Supervisor bits，保留位）
7	D（Dirty，脏位）
6	A（Accessed，访问位）
5	G（Global，全局位）
4	U（User，用户模式位）
3	X（可执行位）
2	W（可写位）
1	R（可读位）
0	V（有效位）

每个字段的具体含义如下：

- **PPN（Physical Page Number）**：位于第53位到第10位，共计44位，表示页表项映射的物理页号（即映射的物理地址）。
- **RSW（Reserved for Supervisor）**：位于第9位和第8位，留给S Mode（Supervisor模式）使用，可用于扩展操作系统的功能。
- **D（Dirty）**：位于第7位，表示该页是否被写入。当D=1时，表示该页已被修改；当D=0时，表示该页未被修改，写入时需要进行置位。
- **A（Accessed）**：位于第6位，表示该页是否被访问过。当A=1时，表示该页被读取或写入。
- **G（Global）**：位于第5位，表示是否为全局页表项。全局页表项在所有地址空间中都可用，不会在上下文切换时被刷新。
- **U（User）**：位于第4位，表示用户模式下的进程是否可以访问该页。当U=1时，用户模式程序可以访问该页；当U=0时，只有S Mode程序可以访问该页。

- **X（Executable）**：位于第3位，表示该页是否可执行。当X=1时，该页允许执行指令；否则，执行指令会导致异常。
- **W（Writable）**：位于第2位，表示该页是否可写。当W=1时，该页允许写操作。
- **R（Readable）**：位于第1位，表示该页是否可读。当R=1时，该页允许读操作。
- **V（Valid）**：位于第0位，表示该页表项是否有效。如果V=0，表示该页表项无效，其他所有位都会被忽略。

关键操作

1. **刷新TLB**：如果页表项的内容被修改，必须刷新TLB（Translation Lookaside Buffer）以确保缓存中的虚拟地址映射被更新。
2. **访问控制**：通过`R`、`W`、`X`位控制页面的读写执行权限，通过`U`位控制用户模式的访问权限。

3. 刷新TLB

刷新TLB的作用是确保虚拟地址到物理地址的映射是最新的和正确的。当页面表中的映射发生变化时，例如由于进程的切换、页表项的更新、内存映射或权限更改，TLB 中可能会缓存过期的或错误的虚拟地址到物理地址的映射。为了避免使用这些过期的映射，必须刷新 TLB。

1. **映射更新**：当内存管理单元（MMU）中的页表项更新时，例如修改了页框号（PPN）或页面权限，如果 TLB 中仍然缓存着旧的映射，CPU 可能会继续使用过时的物理地址进行访问，导致数据错误或权限违规。因此需要刷新 TLB，使其重新加载最新的映射。
2. **进程切换**：不同的进程拥有各自的虚拟地址空间，当进程切换时，TLB 中的映射可能是上一个进程的地址空间，直接使用可能导致访问错误的物理地址。为了确保切换到新进程时映射正确，通常需要刷新 TLB。
3. **内存管理操作**：在系统执行某些内存管理操作（如修改页表、分配或释放内存）时，虚拟地址和物理地址的对应关系发生了改变，必须通过刷新 TLB 来使得这些操作生效。

通过刷新 TLB，系统可以清除过期的映射，使 CPU 在访问虚拟内存时重新查找页表，获取最新的虚拟地址到物理地址的转换，从而避免潜在的错误访问。