练习1：理解first-fit 连续物理内存分配算法

一、物理内存分配过程

在展开物理内存分配流程前，我先对核心数据结构进行澄清：每个物理页由 `struct Page` 表示，`ref` 为引用计数，`flags` 为状态位，`property` 仅在空闲块的头页有效，记录该连续空闲块的页数；`page_link` 是挂入空闲目录的链表节点。空闲目录由 `free_area_t` 管理：`free_list` 为循环双向链表的哨兵，链上 每个节点都对应一个空闲连续块的头页，`nr_free` 统计全部空闲页的总量 。整体设计采用“ 侵入式链表 + 头页记录块大小 ”的方式，把所有空闲连续块组织成一个可高效遍历并支持 O(1) 插入/删除的目录。

根据我的理解，系统启动之后，内存管理走这么几步：

1. 初始化目录（`default_init`）
   这个阶段主要是把空闲目录建出来，即将 `free_list` 设为自环的空链表，`nr_free` 清零。此时系统还不能分配，直到有可用物理页被登记进来。
2. 登记空闲块（`default_init_memmap(base, n)`）
   接着，我们把 `[base, base+n)` 这段合适的连续物理页收编为一个“空闲块”，即清每页的 `flags/ref`，只在头页 `base` 上写 `property=n` 并置 `PG_property`，然后按物理地址顺序把头页节点插入 `free_list`，并将 `nr_free += n`。
3. 分配（`default_alloc_pages(n)`）
   找到空闲块后，我们就可以进行分配，即按 First-Fit 从链表前向后找第一个 `property ≥ n` 的块头 `p`。找到后先把旧块节点从链表摘掉，若块大于需求，则把剩余部分从 `p+n` 起作为新的块头插回原位置，同时 `nr_free -= n`，清 `p` 的 `PG_property`，返回 `p`（代表连续 `n` 页）。
4. 释放（`default_free_pages(base, n)`）
   最后，就是使用完之后的释放了。我们将 `[base, base+n)` 清为“空闲”并把 `base` 设为块头（`property=n`、置位 `PG_property`），按地址有序插入 `free_list`；随后尝试相邻合并，先与左邻相接则合并并删掉被合并节点，再与右邻相接则继续合并；最后 `nr_free += n` 更新全部空闲页的总量。

我们发现其实对于 `default_pmm.c` 这个文件而言，最重要的其实就是这四个函数，那么接下来我将结合具体代码对这几个函数进行更详细的分析。

二、重要函数作用分析

`default_init` —— 建空目录

```
static void default_init(void)
{
    list_init(&free_list);
    nr_free = 0;
}
```

`default_init` 做的就是把“空闲目录”清成一张白纸：调用 `list_init(&free_list);` 把哨兵节点设为 自环的空链表 （此时 `list_empty(&free_list)` 为真），然后 `nr_free = 0;` 将空闲页总数清零。效果是：在没有任何空闲块登记前，`default_alloc_pages` 会先以 `if (n > nr_free) return NULL;` 这一剪枝直接失败——因为目录里没有页可分；必须等 `default_init_memmap(base, n)` 把 `[base, base+n)` 登记为一个空闲块（设置 `base->property=n; SetPageProperty(base);` 并按地址有序插入 `free_list` 同时 `nr_free += n;`），系统才具备后续的首次适配分配能力。换言之，`free_list` 是唯一的“库存目录”，`nr_free` 是唯一的“库存计数”，初始化阶段把两者清空就确保了任何误分配都不会发生。

`default_init_memmap` —— 把一段物理内存登记为一个空闲块

```
static void default_init_memmap(struct Page *base, size_t n) {
    assert(n > 0);
    struct Page *p = base;
    for (; p != base + n; p ++) {
        assert(PageReserved(p));
        p->flags = p->property = 0;
        set_page_ref(p, 0);
    }
    base->property = n;
    SetPageProperty(base);
    nr_free += n;
    if (list_empty(&free_list)) {
        list_add(&free_list, &(base->page_link));
    } else {
        list_entry_t* le = &free_list;
        while ((le = list_next(le)) != &free_list) {
            struct Page* page = le2page(le, page_link);
            if (base < page) {
                list_add_before(le, &(base->page_link));
```

```
                break;
            } else if (list_next(le) == &free_list) {
                list_add(le, &(base->page_link));
            }
        }
    }
}
```

`default_init_memmap(base, n)` 用来把一段连续物理页 `[base, base+n)` 注册为一个空闲块。它先对这段里的每一页做“入库净化”：清空 `flags/property/ref`；随后仅在头页 `base` 上写入 `property = n` 并置位 `PG_property`，表示“从这里起连续有 n 页空闲”。接着把 `nr_free += n`，更新全局空闲页统计。最后，将 `base->page_link` 按物理地址有序插入 `free_list`（空表则直接插入；否则用 `list_next` 线性扫描，找到第一个地址大于 `base` 的块头，用 `list_add_before` 插入；若未找到，则用 `list_add` 接在链尾）。之所以强调“有序插入”，是为了后续释放/合并时能通过 `prev/next` O(1) 判断相邻关系：释放某段后，只需检查它在链表中的左邻与右邻是否与当前块首尾相接，即可完成“先左后右”的增量合并，而无需全表查找。

`default_alloc_pages` —— 首次适配 + 切分

```
static struct Page * default_alloc_pages(size_t n) {
    assert(n > 0);
    if (n > nr_free) {
        return NULL;
    }
    struct Page *page = NULL;
    list_entry_t *le = &free_list;
    while ((le = list_next(le)) != &free_list) {
        struct Page *p = le2page(le, page_link);
        if (p->property >= n) {
            page = p;
            break;
        }
    }
    if (page != NULL) {
        list_entry_t* prev = list_prev(&(page->page_link));
        list_del(&(page->page_link));
        if (page->property > n) {
            struct Page *p = page + n;
            p->property = page->property - n;
```

```
            SetPageProperty(p);
            list_add(prev, &(p->page_link));
        }
        nr_free -= n;
        ClearPageProperty(page);
    }
    return page;
}
```

`default_alloc_pages(n)` 的工作流程是：先做 剪枝 ，如果 `n > nr_free` 直接返回 `NULL`；否则从 `free_list` 的哨兵开始，用 `while ((le=list_next(le)) != &free_list)` 顺序遍历，每次用 `le2page(le, page_link)` 取到块头页 `p`，按 First-Fit 规则选中第一个满足 `p->property >= n` 的空闲块并记为 `page`。找到后先保存其链表前驱 `prev = list_prev(&(page->page_link))`，再 `list_del(&(page->page_link))` 把旧块从目录摘下；如果原块大于需求（`page->property > n`），就把剩余部分从 `page+n` 起作为 新的块头：`p = page + n; p->property = page->property - n; SetPageProperty(p);`，并用 `list_add(prev, &(p->page_link))` 就地插回，保证 `free_list` 仍按 物理地址有序 。随后更新全局统计 `nr_free -= n`，并对返回的头页 `page` 执行 `ClearPageProperty(page)`（它已不再是空闲块头），最后返回 `page`，表示成功分配了一段 连续的 n 个页 。整个过程维持三条不变式：`free_list` 始终按地址递增、链上每个节点对应一个空闲连续块的头页、`nr_free` 等于所有空闲块页数之和。

`default_free_pages` —— 插回 + 左右合并

```
static void default_free_pages(struct Page *base, size_t n) {
    assert(n > 0);
    struct Page *p = base;
    for (; p != base + n; p ++) {
        assert(!PageReserved(p) && !PageProperty(p));
        p->flags = 0;
        set_page_ref(p, 0);
    }
    base->property = n;
    SetPageProperty(base);
    nr_free += n;

    if (list_empty(&free_list)) {
        list_add(&free_list, &(base->page_link));
    } else {
        list_entry_t* le = &free_list;
```

```
        while ((le = list_next(le)) != &free_list) {
            struct Page* page = le2page(le, page_link);
            if (base < page) {
                list_add_before(le, &(base->page_link));
                break;
            } else if (list_next(le) == &free_list) {
                list_add(le, &(base->page_link));
            }
        }
    }

    list_entry_t* le = list_prev(&(base->page_link));
    if (le != &free_list) {
        p = le2page(le, page_link);
        if (p + p->property == base) {
            p->property += base->property;
            ClearPageProperty(base);
            list_del(&(base->page_link));
            base = p;
        }
    }

    le = list_next(&(base->page_link));
    if (le != &free_list) {
        p = le2page(le, page_link);
        if (base + base->property == p) {
            base->property += p->property;
            ClearPageProperty(p);
            list_del(&(p->page_link));
        }
    }
}
```

`default_free_pages(base, n)` 的流程是：先将 `[base, base+n)` 这段页清为“空闲”，把 `base` 标为 新块头 （`property=n`、置位 `PG_property`），并将 `nr_free += n`；随后按物理地址有序把 `base->page_link` 插入 `free_list`。插入后立即尝试合并： 先看左邻，若其末尾与 `base` 的开头相接（`p + p->property == base`），则把当前块并入左邻 `p`，删除 `base` 的链表节点，并把 `base=p` 作为新的块头； 再看右邻，若（可能已扩大的）`base` 的末尾与右邻 `p` 的开头相接，则继续把右邻并入 `base` 并删除其节点。整个过程中，被合并的一侧要清除 `PG_property`。完成后，`free_list` 继续保持 按地址有序，目录上每个

节点都代表一段 连续空闲块的头页 ， `property` 仍只记录在头页上，而 `nr_free` 与总空闲页数一致（这里在入库时一次性 `+= n`，合并不再改变页总数）。

First Fit 算法优化思路

在不改变“从前往后找，遇到第一块够大就用”的前提下，first-fit 也能做得更顺手：其一，用 **next-fit** 给它加个“游标”，每次从上次命中的位置继续扫，命中规则仍是 first-fit，但少了无谓回头路；其二，做个 **按大小分桶的 first-fit** ，先把空闲块按容量分到几个链里，分配时先选合适的桶，再在桶内用 first-fit 拿到第一个能用的块，既快又更少撕裂大块；其三，反过来也可以 **优先从“大桶”开始再 first-fit** ，把消耗集中在大块上，给中等请求留出更稳的空间；其四，换一套更规整的底座——**伙伴系统**按 2 的幂分阶管理，但每个阶里的选择依旧用 first-fit，这样合并/拆分是常数开销，碎片也更可控。在后面的练习中，我们会就此有选择性地进行一系列优化。

练习2：实现 Best-Fit 连续物理内存分配算法

从实现思路上，`First Fit` 是从空闲列表的头部开始遍历，找到第一个满足分配需求的空闲块，随即遍历停止；而 `Best Fit` 则遍历整个空闲列表，找到最小的、但仍然满足分配需求的空闲块。因此，结合上面我们对 `First Fit` 实现流程的理解，我们会发现我们其实只需要修改 `default_alloc_pages(n)` 函数中找到那个所需要块的代码即可，具体修改后的代码如下：

```
while ((le = list_next(le)) != &free_list) {
   struct Page *p = le2page(le, page_link);
    if (p->property >= n && p->property < min_size) {
       page = p, min_size = p->property;
    }
}
```

可以看到，这里把 `default_alloc_pages(n)` 中“选块”的那一段改成 Best-Fit ：仍然从 `free_list` 头往后遍历，但不再遇到第一个就停，而是 扫描整条空闲链 ，用 `min_size` 记录当前找到的最小、且 ≥ n 的块；遍历中每遇到 `p->property >= n` 且 `p->property < min_size` 的块，就把 `page` 更新为 `p`、同时更新 `min_size`。最终循环结束时，`page` 就指向“ 最小但足以满足 n ”的空闲块。其余流程（从链表摘下该块、必要时切分、把剩余部分就地回插、更新 `nr_free`）保持不变，只是把 First-Fit 的“第一块” 替换成 Best-Fit 的“最合适那块” ，以此更好地保护大块、降低外部碎片。
写完代码之后尝试验证是否正确，我们使用 `make grade` 这个命令行，首先我们来观察一下当我们输入 `make grade` 之后发生了什么，下面是验证过程的详细说明：

1. 清理项目

   make grade 首先调用 clean 目标，其作用是清理项目，避免旧文件影响编译结果。这个步骤通过删除所有中间文件（如 .o 文件）和之前生成的目录（如 obj 和 bin），确保了后续编译的纯净环境。

```
clean:
    $(V)$(RM) $(GRADE_GDB_IN) $(GRADE_QEMU_OUT) cscope* tags
    -$(RM) -r $(OBJDIR) $(BINDIR)
```

2. 编译内核

   清理完成后，make grade 会重新编译内核。这一步的作用是将链接好的内核 ELF 文件（$(kernel)）转换为裸机可执行的二进制镜像文件 ucore.img。这是通过 objcopy 工具实现的，生成的 ucore.img 将用于后续在 QEMU 模拟器中加载和运行。

```
$(UCOREIMG): $(kernel)
    $(OBJCOPY) $(kernel) --strip-all -O binary $@
```

3. 运行测试脚本

   内核编译完成后，make grade 会执行核心的测试脚本 grade.sh。该脚本的作用是启动 QEMU 模拟器加载 ucore.img，并捕获内核运行时的所有输出。它通过内部定义的 quick_check 等函数，使用正则表达式匹配 QEMU 的输出流，从而自动化地验证实验结果是否符合预期。

```
pts=5
quick_check 'check physical_memory_map_information' \
    'memory management: best_fit_pmm_manager' \
    '  memory: 0x0000000008000000, [0x0000000080000000,
0x0000000087ffffff].'

pts=20
quick_check 'check_best_fit' \
    'check_alloc_page() succeeded!' \
    'satp virtual address: 0xffffffffc0204000' \
    'satp physical address: 0x0000000080204000'
```

4. 验证输出

   grade.sh 脚本会捕获 QEMU 的输出，并进行一系列验证。它会检查内存管理器是否已正确初始化为 best_fit_pmm_manager，同时确认页表的虚拟地址和物理地址是否按预期输出。最后，它还会通过检查 check_alloc_page() succeeded! 的输出来验证 best_fit 内存分配算法本身的功能是否正确。

随后我们就可以得出我们的验证结果，可以看到结果显示正确。这样我们就实现了 `Best Fit` 。

Challenge1 : buddy_system_pmm实现

buddy_system 原理简述

buddy_system 与 First Fit 和 Best Fit 不同，前者将所有的页框都划分为 2 的次幂，无论申请多少物理页，都会划分出最小的大于其申请数量的 2 次幂页框，而后者是申请多少就给出多少物理页。

buddy_system 同时使用了链表和类似二叉树的结构，使用 $\log n$ 个链表，其中第 i 个链表储存所有大小为 2^i 的页框。若需要申请一个大小为 2^n 的页框，首先在对应的链表上查找，若不存在这个大小的页框，那就转而申请一个大小为 2^{n+1} 的页框，将其分解成两个大小为 2^n 的页框，并返回其中一个即可。若将 2^{n+1} 的页框看作父亲，将两个 2^n 的节点看作孩子，那么 buddy_system 的所有可能的页框会形成一个类似二叉树的结构，所有可用的页框中，相同深度的页框具有相同的大小，因此排列在同一个链表上（需要注意的是buddy_system 并不要求一个链表上的页框是按照物理页地址排序的）。在释放内存时同理，需要找到释放页框的兄弟节点，若兄弟节点存在的话，将这两个节点合并，同时更新链表。

这样的好处在于，若使用 First Fit 或 Best Fit，申请页框需要遍历所有的页框，而 buddy_system 则只需要遍历至多 $\log n$ 个页框。同理，buddy_system 释放页框也可能需要遍历 $\log n$ 个页框，而 First Fit 或 Best Fit 则需要遍历所有页框才能找到插入的位置。同时，由于我们在 buddy_system 中，我们申请的页框大小均为 2 的次幂，方便维护的同时不会产生太多内存碎片，而 First Fit 和 Best Fit 则容易产生大量无序的内存碎片。

buddy_system 的位运算实现

在给出的教程中，建立出了一棵完全二叉树来维护页框，然而这样需要申请大小为 $2n$ 的结构体数组，同时维护的物理页数量也必须恰好是 2 的次幂。这里我们使用一种更简洁的位运算方法来完成占用空间更少、运行速度也更快的 buddy_system。

教程中使用完全二叉树的目的是帮助我们寻找页框的兄弟节点和父亲节点，从而完成一次合并。现在我们不使用完全二叉树来完成这样的寻找。首先我们将所有 pmm 负责的所有物理页从 0 开始编号，假设将所有的页框大小都划分为 2^i ，那么从左到右所有的页框包含的物理页依次为 $[0, 2^i), [2^i, 2 \times 2^i), [2 \times 2^i, 3 \times 2^i) \dots$ 。此时如果按照 buddy_system 的规则将所有页框都合并为 2^{i+1} 大的页框，则会将
$[2k \times 2^i, (2k+1) \times 2^i), [(2k+1) \times 2^i, (2k+2) \times 2^i)$ 合并为 $[k \times 2^{i+1}, (k+1) \times 2^{i+1})$ 。可以注意到，对于起始物理页编号为 id ，大小为 2^i 的页框，其兄弟节点就是起始物理页编号为 $id \oplus 2^i$ ，大小为 2^i 的页框，其中 $\oplus$ 表示按位异或。其父亲节点就是起始物理页编号为 $min(id, id \oplus 2^i)$，大小为 2^{i+1} 的页框。

有了这一性质之后，我们就可以完全依赖 lab2 中的 `Page` 结构体而不需要创建一颗完全二叉树来实现 buddy_system 。

C语言代码实现

pmm 初始化

buddy_system使用 $\log n$ 个链表维护 $\log n$ 中不同大小的页，因此在初始化时需要将这写链表初始化。由于我现在无法使用 `alloc`（先有鸡？还是先有蛋？），我也没有掌握更高深的分配内存技巧，我定义了一个大小为 15 的free_area_t数组来维护链表。

其中 `page_base` 指针用于记录第一个物理页的地址，方便计算物理页的 id。

```
static free_area_t free_area[15];
static struct Page *page_base = NULL;
static uint32_t nr_free = 0;

#define free_list(x) (free_area[x].free_list)
#define nr_free(x) (free_area[x].nr_free)

static void
buddy_system_init(void) {
    for(int i = 0; i < 15; ++i) {
        list_init(&free_list(i));
        nr_free(i) = 0;
    }
}
```

初始化内存

初始化内存时，我们将 n 用二进制表示，每出现一个1就分配一个页框，例如 $13 = 1101_2$ ，则分配三个大小分别为 $8, 4, 1$ 的页框，$[0000_2, 1000_2), [1000_2, 1100_2), [1100_2, 1101_2)$ 。

```
static void
buddy_system_init_memmap(struct Page *base, size_t n) {
    assert(n > 0);
    struct Page *p = base;
    page_base = base;
    for (; p != base + n; p ++) {
        assert(PageReserved(p));
        p->flags = p->property = 0;
        set_page_ref(p, 0);
```

```
    }
    nr_free += n;
    for(int i = 0; i < 15; ++i) if((n >> i) & 1) {
        struct Page *q = base + (n -= 1 << i);
        q->property = 1 << i;
        SetPageProperty(q);
        list_add(&free_list(i), &(q->page_link));
        ++ nr_free(i);
    }
}
```

分配页框

分配页框时，首先要找到最小的大于 n 的页框，之后将这个页框不断分裂，知道其恰好大于 n 时，返回这个页框。

```
static struct Page *
buddy_system_alloc_pages(size_t n) {
    assert(n > 0);
    if (n > nr_free) {
        return NULL;
    }
    struct Page *page = NULL;
    uint8_t log = 0;
    for(int i = 0; i < 15; ++i) {
        if(n <= (1 << i) && nr_free(i)) {
            page = le2page(list_next(&free_list(log = i)),
page_link);
            break;
        }
    }
    if (page != NULL) {
        list_del(&(page->page_link));
        --nr_free(log);
        while(log && n <= (1 << (log - 1))) {
            ++nr_free(--log);
            struct Page *p = page + (1 << log);
            p->property = (1 << log);
            p->flags = 0;
            set_page_ref(p, 0);
            SetPageProperty(p);
            list_add_before(&free_list(log), &(p->page_link));
        }
```

```
        nr_free -= 1 << log;
        page->property = 1 << log;
        ClearPageProperty(page);
    }
    return page;
}
```

释放页框

释放页框时，不断寻找兄弟节点向上合并即可。

```
static void
buddy_system_free_pages(struct Page *base, size_t n) {
    assert(n > 0);
    size_t page_id = base - page_base;
    base->property = n;
    SetPageProperty(base);
    nr_free += n;

    for(int i = 0; i < 15; ++i) if((n >> i) & 1){
        size_t buddy_page_id = page_id ^ (1 << i);
        struct Page *buddy_page = page_base + buddy_page_id;
        if(PageProperty(buddy_page) && buddy_page->property ==
base->property) {
            if(buddy_page_id < page_id)
                page_id = buddy_page_id;
            base = page_base + page_id;
            base->property = 2 << i;
            list_del(&(buddy_page->page_link));
            --nr_free(i);
            ClearPageProperty(base + (1 << i));
            n <<= 1;
        } else {
            list_add_before(&free_list(i), &(base->page_link));
            ++nr_free(i);
            break;
        }
    }
}
```

Challenge 2 : A simplified slub pmm manager

slub 原理简述

在我们使用 buddy_system 管理内存时，申请内存的最小单位为页。然而许多数据结构所需要的内存远小于一个页，这将导致两种可能的情况，一是一个很小的数据结构占据了一整页的内存，导致内存浪费严重，二是软件必须自己实现一个小的内存管理系统，专注于管理一个页内的内存使用情况，既不方便、也不安全。

slub pmm manager 应运而生。slub 使用 buddy_system 作为低级的分配器，若要申请页，则直接使用buddy_system 进行操作，若要大量申请 比页更小的内存，则首先申请一个对应大小的 `kmem_cache` ，再通过 `kmem_cache` 进行申请。

slub 的核心数据结构为 `kmem_cache` 和 `slab` 。每个 `slab` 中包含若干个页（本次实验为了简化，一个 `slab` 中只包含一个页），一个 `slab` 中的内存被等分为若干份，每份表示一个 `objects` ，同一个 `slab` 中所有的 `object` 大小相同。每个 `kmem_cache` 管理若干个 `slab` ，同一个 `kmem_cache` 管理的 `slab` 中的 `object` 大小均相同。

由于本次实验不需要并发控制，因此不需要对页面加锁。

一个 kmem_cache 如何管理 slab

首先我们知道一个 `kmem_cache` 管理的所有 `object` 大小都是相同的，因此 `kmem_cache` 结构体中首先要包含 `object` 的大小。

slub 系统的设计与计算机硬件结构是紧密结合的。现代计算机中常出现多个 CPU 的情况，此时我们需要使用 NUMA 进行内存的管理，一个 NUMA node 中包含 若干CPU核心和内存，若我们想要给一个 CPU 分配新的内存，首先应该在当前 CPU 变量区中找是否有空余的 `object` ，若没有，再去CPU 对应的 NUMA node 中寻找。

因此 `kmem_cache` 结构体需要管理两个结构体，分别是 `kmem_cache_cpu` 和 `kmem_cache_node` ，其中 `kmem_cache_cpu` 管理 `per-CPU area` 中所有的 slab，而 `kmem_cache_node` ，管理 NUMA node 中的所有 `slab` 。每个 CPU 有一个对应的 `kmem_cache_cpu` ，每个 NUMA node 有一个对应的 `kmem_cache_node` 。若两个 CPU 处于同一个 NUMA node 中，则他们在申请 NUMA node 中的内存时，都会从这个 NUMA node 对应的 `kmem_cache_node` 中申请。（由于环境限制，本次实验中假设只有一个CPU和一个NUMA node）。

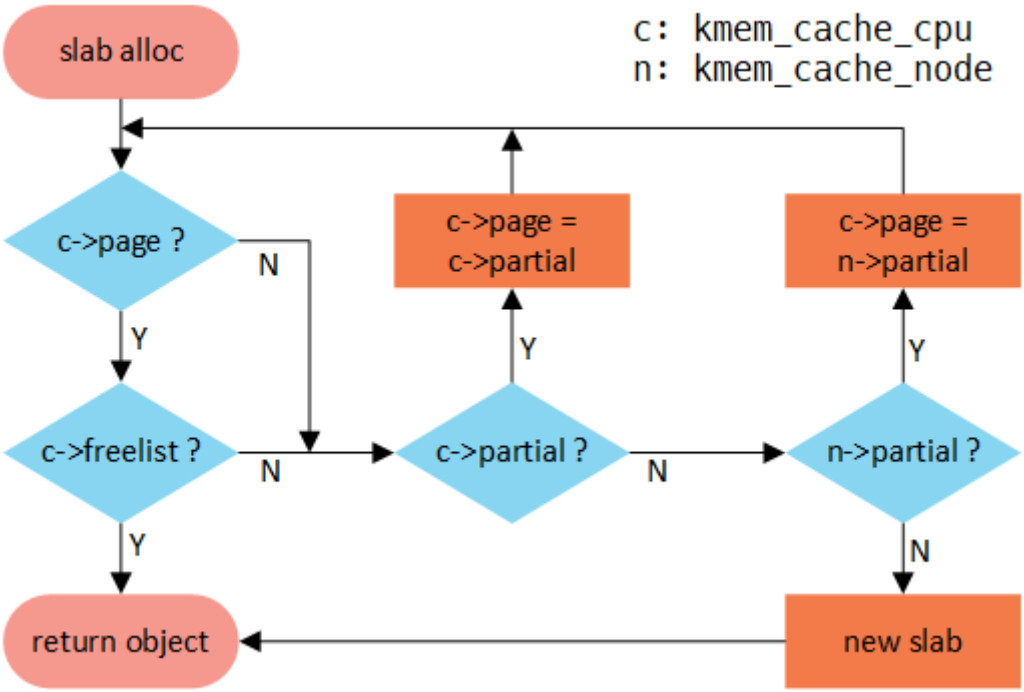
`kmem_cache_cpu` 结构体中需要管理一个 `slab` ，表示当前活跃的 `slab` ，申请新内存时优先从这里申请。需要包含一个 `slab_list` ，表示部分满的 `slab` ，若当前活跃的 `slab` 满了，将从这里获取新的 `slab` 。需要维护一个整数表示当前 `slab_list` 中可用的 `object` 数量，我们不希望一个 CPU 上有过多的 `object` ，若这个数超过一个定值（本次实验中这

个数被设置为常数），会将 `slab_list` 上的 `slab` 还给 `kmem_cache_node` 管理。

`kmem_cache_node` 结构体中需要包含一个 `slab_list` 表示这个 NUMA node 管理的所有 `slab` 。需要包含一个 `nr_partial` 表示 `slab_list` 的大小，若 `kmem_cache_node` 管理了过多的 `slab` （本次实验中这个数被设置为定值），则会将空的 `slab` 回收。

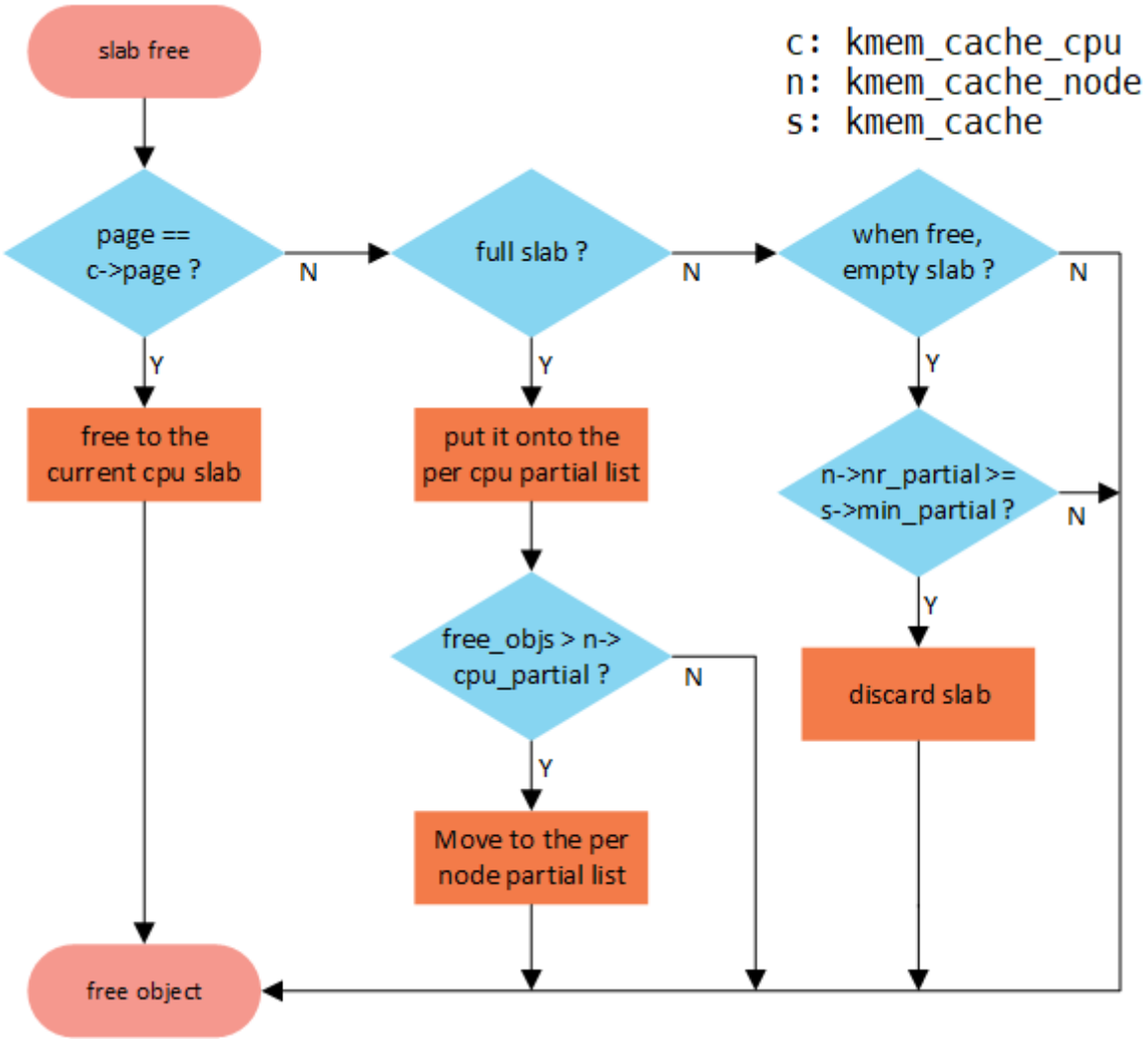
如何从 kmem_cache 申请一个 object

首先我们需要从对应 CPU 的 `kmem_cache_cpu` 中找到当前活跃的 `slab`，如果当前有活跃的 `slab`，且页不满，则直接从这个 `slab` 中申请即可。若不存在活跃的`slab` ，则直接从 buddy_system 中申请一个页，并将其初始化为一个 `slab`。若活跃的 `slab` 已经满了，就从 `kmem_cache_cpu->partial` 中寻找一个 `slab`，将其作为新的活跃 `slab` 。若 `kmem_cache_cpu->partial` 中也没有 `slab` 了，就从 CPU 对应 NUMA node 的 `kmem_cache_node->partial` 中找一个 `slab` 作为新的活跃 `slab`，若 `kmem_cache_node` 中也找不到合适的 `slab` ，就再从 buddy_system 中申请一个新的页将其作为活跃 `slab`.



如何释放一个 object

首先我们要观察释放的 `object` 所在的 `slab` 位于哪里，若位于当前 CPU 的 `slab` 上，直接施放即可。若位于一个满的 `slab` 中，则需要将这个 `slab` 放到 `kmem_cache_cpu->partial` 中，若当前 `kmem_cache_cpu->partial` 上已经包含了过多的 `object` ，则需要将 `kmem_cache_cpu->partial` 上的所有 `slab` 转移到 `kmem_cache_node->partial` 上，再将这个页加入。若释放的 `object` 在一个空的 `slab` 上，且这个 `slab` 位于 `kmem_cache_node->partial` ，则需要判断 `kmem_cache_node` 是否包含了过多的 `slab` ，若是，则可以将这个 `slab` 释放掉。



如何创建一个新的 kmem_cache

简化后的 `slab` 只需要分别实现对 `kmem_cache`，`kmem_cache->kmem_cache_cpu`，`kmem_cache->kmem_cache_node` 的初始化即可。问题来了，我们要如何分配这三个数据结构的内存？若直接申请一个页，未免过于浪费。但如果用 slub 进行分配，则必然面临鸡生蛋还是蛋生鸡的问题。我们可以在内核初始化的时候先在内核中分配三只`创世鸡`，分别用于分配 `kmem_cache` 、`kmem_cache_node` 和 `kmem_cache_cpu` 。这样在创建新的 `kmem_cache` 时就可以直接用 slub 分配内存。

若对于将要分配的 `kmem_cache` ，已经存在大小相似的 `kmem_cache` 可以完成其功能，则可以这两个 `kmem_cache` 进行合并。但是实验中为了简化跳过了这一步骤。

如何创建一个新的 slab

创建一个 `slab` 需要初始化一个结构体，用于储存 `slab` 内的所有信息，同时需要从 buddy_system 中申请一个页，将其划分成若干 `object` 。后者是简单的，但对于前者来说，我们要将这个结构体储存在哪里呢？如果像 `kmem_cache` 一样，使用一个 `创世鸡` 来完成 `slab` 的页面分配，则可能会出现无限递归的情况，即从 slub 中分配一个 `object` 需要

创建一个新的 `slab`，而创建一个新的 `slab` 需要向 slub 申请一个 `object`。

因此，在本次实验中，我将 `slab` 数据结构放在了`slab` 所在页的最前面，后面的部分才用来储存 `object`。

如何删除一个 kmem_cache

删除 `kmem_cache` 则相对简单，确认 `kmem_cache` 中没有未被释放的 `object` 后，就可以释放 `kmem_cache`。

C 语言代码实现

设计需要用到的结构体

需要用到的结构体如下所示：

```
typedef struct free_object {
    struct free_object *next;
} free_object_t;

struct slab {
    free_object_t *free_obj;
    struct Page *page;
    list_entry_t slab_list;
    size_t inuse;
    size_t nr_free_objs;
};

struct kmem_cache_node {
    list_entry_t slab_link;
    size_t nr_partial;
} ;

struct kmem_cache_cpu {
    list_entry_t partial_link;
    struct slab *page;
    size_t freeobjs;
} ;

struct kmem_cache {
    size_t size;
```

```
    size_t align;
    size_t min_partial;
    size_t cpu_partial;
    size_t inuse;
    char *name;
    list_entry_t cache_link;
    struct kmem_cache_node *kmem_node;
    struct kmem_cache_cpu *kmem_cpu;
} ;
```

对 slub 进行初始化

首先要对 buddy_system 进行初始化，之后要对三只 `创世鸡` 进行初始化，他们将负责 `kmem_cache` 的内存分配。

```
static void
kmem_cache_node_init(struct kmem_cache_node *kmem_node) {
    kmem_node->nr_partial = 0;
    list_init(&(kmem_node->slab_link));
}

static void
kmem_cache_cpu_init(struct kmem_cache_cpu *kmem_cpu) {
    kmem_cpu->freeobjs = 0;
    kmem_cpu->page=NULL;
    list_init(&(kmem_cpu->partial_link));
}

static void
kmem_cache_init(struct kmem_cache *kmem, char *name, size_t size) {
    kmem->cpu_partial = 128;
    kmem->min_partial = 8;
    kmem->inuse = 0;
    kmem->size = size;
    kmem->name = name;
    kmem->align = ROUNDUP(size + sizeof(free_object_t),
sizeof(char*));
    list_init(&(kmem->cache_link));
    list_add_before(&kmem_cache_list, &(kmem->cache_link));
    kmem_cache_cpu_init(kmem->kmem_cpu);
    kmem_cache_node_init(kmem->kmem_node);
}
```

```
static void
slub_init(void) {
    lower_pmm_manager = &buddy_system_pmm_manager;
    lower_pmm_manager->init();
    list_init(&kmem_cache_list);

    kmem_cache_allocater = &kmem_cache;
    kmem_cache_cpu_allocater = &kmem_cpu;
    kmem_cache_node_allocater = &kmem_node;

    kmem_cache.kmem_cpu = &kmem_cache_cpu;
    kmem_cache.kmem_node = &kmem_cache_node;
    kmem_cpu.kmem_cpu = &kmem_cpu_cpu;
    kmem_cpu.kmem_node = &kmem_cpu_node;
    kmem_node.kmem_cpu = &kmem_node_cpu;
    kmem_node.kmem_node = &kmem_node_node;
    kmem_cache_init(&kmem_cache, "kmem_cache_allocater",
sizeof(struct kmem_cache));
    kmem_cache_init(&kmem_cpu, "kmem_cache_cpu_allocater",
sizeof(struct kmem_cache_cpu));
    kmem_cache_init(&kmem_node, "kmem_cache_node_allocater",
sizeof(struct kmem_cache_node));
}
```

创建一个新的 `slab`

将页面最开始的部分用于存储 `slab` 结构体，之后的部分用于初始化 `object` 。

```
static struct slab*
get_new_slab(struct Page *page, size_t align) {
    uintptr_t page_pa = page2pa(page);
    uintptr_t page_va = (uintptr_t)KADDR(page_pa);
    struct slab *new_slab = (struct slab*)page_va;
    new_slab->page = page;
    new_slab->inuse = 0;
    new_slab->nr_free_objs = (PGSIZE - sizeof(struct slab)) /
align;
    list_init(&(new_slab->slab_list));
    page_va += sizeof(struct slab);
    memset((char*)page_va, 0, PGSIZE - sizeof(struct slab));
    for(int i = 0; i < new_slab->nr_free_objs; ++i) {
        if(i == new_slab->nr_free_objs - 1)
```

```
            ((free_object_t*)(page_va + i * align))->next = NULL;
        else
            ((free_object_t*)(page_va + i * align))->next =
(free_object_t*)(page_va + (i + 1) * align);
    }
    new_slab->free_obj = (free_object_t*)(page_va);
    return new_slab;
}
```

分配一个 object

```
void *kmem_cache_alloc(struct kmem_cache *kc) {
    struct slab* retslab = NULL;
    struct kmem_cache_cpu *kmem_cpu = kc->kmem_cpu;
    struct kmem_cache_node *kmem_node = kc->kmem_node;
    if(kmem_cpu->page == NULL) { // 初始时，kmem_cpu上没有slab
        struct Page *new_page = lower_pmm_manager->alloc_page();
        kmem_cpu->page = get_new_slab(new_page, kc->align);
        kmem_cpu->freeobjs += kmem_cpu->page->nr_free_objs;
        retslab = kmem_cpu->page;
    } else if(kmem_cpu->page->free_obj != NULL) { // kmem_cpu上的
slab有obj
        retslab = kmem_cpu->page;
    } else if(!list_empty(&(kmem_cpu->partial_link))) { // kmem_cpu
上的slab无空余obj，从cpu_partial_list上找
        kmem_cpu->page = le2slab(list_next(&(kmem_cpu-
>partial_link)), slab_list);
        list_del(&(kmem_cpu->page->slab_list));
        kmem_cpu->freeobjs -= kmem_cpu->page->nr_free_objs;
        retslab = kmem_cpu->page;
    } else if(!list_empty(&(kmem_node->slab_link))) { // kmem_cpu的
partial_list上也没有空余slab，去kmem_node->partial中寻找
        kmem_cpu->page = le2slab(list_next(&(kmem_node-
>slab_link)), slab_list);
        list_del(&(kmem_cpu->page->slab_list));
        kmem_node->nr_partial--;
        retslab = kmem_cpu->page;
    } else { // kmem_node上也没有了，重新从 buddy 中申请一页
        struct Page *new_page = lower_pmm_manager->alloc_page();
        kmem_cpu->page = get_new_slab(new_page, kc->align);
        kmem_cpu->freeobjs += kmem_cpu->page->nr_free_objs;
        retslab = kmem_cpu->page;
```

```
    }
    if(retslab == NULL) return NULL;
    uintptr_t ret = (uintptr_t)(retslab->free_obj);
    retslab->nr_free_objs--;
    retslab->inuse++;
    retslab->free_obj = retslab->free_obj->next;
    kc->inuse++;
    return (char*)(ret + sizeof(free_object_t));
}
```

释放一个 `object`

```
void kmem_cache_free(struct kmem_cache *kc, void *o) {
    struct slab *slab = (struct slab*)ROUNDDOWN(o, PGSIZE);
    free_object_t *obj = (free_object_t*)((uintptr_t)o -
sizeof(free_object_t));
    struct kmem_cache_cpu *kmem_cpu = kc->kmem_cpu;
    struct kmem_cache_node *kmem_node = kc->kmem_node;
    obj->next = slab->free_obj;
    slab->free_obj = obj;
    slab->nr_free_objs++;
    slab->inuse--;
    kc->inuse--;
    if(slab->nr_free_objs == 1) { //原先是满的slab
        if(kmem_cpu->freeobjs > kc->cpu_partial) { //清空 cpu
partial，将slab加入cpu partial
            for(list_entry_t *le; !list_empty(&(kmem_cpu-
>partial_link)); ) {
                le = list_next(&(kmem_cpu->partial_link));
                kmem_cpu->freeobjs-=le2slab(le, slab_list)-
>nr_free_objs;
                list_del(le);
                list_add(&(kmem_node->slab_link), le);
                kmem_node->nr_partial++;
            }
        } else { //直接将slab加入cpu partial
            list_add(&(kmem_cpu->partial_link), &(slab-
>slab_list));
        }
        kmem_cpu->freeobjs++;
    } else if(slab->inuse == 0 && kmem_node->nr_partial > kc-
>min_partial) {
```

```
        bool flag = 0; //检测slab是否属于kmem_node，若属于，则需要将这个
slabfree掉。
        for(list_entry_t *le = list_next(&(kmem_node->slab_link));
le != &(kmem_node->slab_link); le = list_next(le))
            if(le == &(slab->slab_list)) { flag = 1; break; }
        if(!flag) return;
        list_del(&(slab->slab_list));
        lower_pmm_manager->free_page(slab->page);
    }
}
```

创建一个 kmem_cache

分配好内存后调用初始化函数即可。

```
struct kmem_cache *kmem_cache_create(char *name, size_t size) {
    struct kmem_cache *new_kmem_cache =
kmem_cache_alloc(kmem_cache_allocater);
    new_kmem_cache->kmem_cpu =
kmem_cache_alloc(kmem_cache_cpu_allocater);
    new_kmem_cache->kmem_node =
kmem_cache_alloc(kmem_cache_node_allocater);
    kmem_cache_init(new_kmem_cache, name, size);
    return new_kmem_cache;
}
```

释放一个 kmem_cache

判断是否存在未释放的 `object` ，若存在则 panic ，否则依次释放。

```
void kmem_cache_destroy(struct kmem_cache *kc) {
    struct kmem_cache_cpu *kmem_cpu = kc->kmem_cpu;
    struct kmem_cache_node *kmem_node = kc->kmem_node;
    assert(kc->inuse == 0);
    if(kmem_cpu->page != NULL) {
        assert(kmem_cpu->page->inuse == 0);
        for(list_entry_t *le = list_next(&(kmem_cpu-
>partial_link)); le != &(kmem_cpu->partial_link); le =
list_next(le))
            assert(le2slab(le, slab_list)->inuse==0);
```

```
        for(list_entry_t *le = list_next(&(kmem_node->slab_link));
le != &(kmem_node->slab_link); le = list_next(le))
            assert(le2slab(le, slab_list)->inuse==0);
        for(list_entry_t *le; !list_empty(&(kmem_cpu-
>partial_link)); ) {
            le = list_next(&(kmem_cpu->partial_link));
            lower_pmm_manager->free_page(le2slab(le, slab_list)-
>page);
        }
        for(list_entry_t *le; !list_empty(&(kmem_node->slab_link));
) {
            le = list_next(&(kmem_node->slab_link));
            lower_pmm_manager->free_page(le2slab(le, slab_list)-
>page);
        }
        lower_pmm_manager->free_page(kmem_cpu->page->page);
    }
    list_del(&(kc->cache_link));
    kmem_cache_free(kmem_cache_node_allocater, kmem_cpu);
    kmem_cache_free(kmem_cache_cpu_allocater, kmem_cpu);
    kmem_cache_free(kmem_cache_allocater, kc);
}
```

Challenge3：硬件的可用物理内存范围的获取方法（思考题）

现代计算机使用**ACPI** 标准。在启动过程中，BIOS（或UEFI）会在内存中创建一个名为**ACPI表**的数据结构。这些表格包含了关于系统硬件的详细信息，其中：

- **SRAT** 表：描述处理器和内存的亲和性。
- **SLIT** 表：描述内存访问延迟。
- **DMAR** 表：描述用于虚拟化的IOMMU重映射。

BIOS中断机制与数据结构

1. INT 15h E820h中断规范

物理内存探测通过调用参数为e820h的INT 15h BIOS中断实现，该中断使用系统内存映射地址描述符格式来表示物理内存布局。调用时需设置以下参数：`eax`寄存器为`0xE820`功能号，`edx`寄存器为`0x534D4150`签名标识（即"SMAP"），`ebx`寄存器作为续传标识（0表示开始，非0表示继续），`ecx`寄存器设置为20字节的缓冲区大小，`es:di`指向保存地址范围描述符的缓冲区。

BIOS中断返回后，每个内存段描述符包含8字节基地址、8字节内存块长度和4字节内存类型。内存类型主要包括：01h（可用内存）、02h（保留内存）、03h（ACPI可回收内存）和04h（ACPI NVS内存）。通过CF标志位判断调用是否成功，`ebx`寄存器作为续传值（0表示结束，非0继续），实现渐进式内存映射探测。

2. 内存描述符数据结构

探测结果按照`struct e820map`数据结构进行组织，该结构包含`nr_map`字段记录内存段数量，以及`map`数组存储各个内存段描述符。每个描述符包含8字节的`addr`（基地址）、8字节的`size`（内存块长度）和4字节的`type`（内存类型），从`0x8004`地址开始连续存储，`0x8000`处存储总段数，便于后续解析和处理。

3. 探测执行流程

物理内存探测在`bootasm.S`中实现，首先初始化阶段将`0x8000`处的`nr_map`计数器清零，初始化`ebx`续传标识为0，并设置`di`指向`0x8004`作为描述符存储起始地址。接着进入循环探测阶段，设置`eax`为`0xE820`功能号，`ecx`为20字节描述符大小，`edx`为"SMAP"签名后调用INT 15h中断。

中断返回后通过检查CF标志位判断是否成功：若失败则设置错误码`12345`并退出探测；若成功则移动缓冲区指针（`di`增加20）、递增内存段计数，并检查`ebx`是否为0决定是否继续探测。这种续传机制确保能够完整枚举系统中的所有内存段，为ucore操作系统的内存管理提供准确的物理内存布局信息。

其他方法

1.设备树

在 ARM、RISC-V 等嵌入式或开放平台上，**设备树** 是描述硬件配置的标准方式。工作原理是引导程序（如 U-Boot）会将一个编译好的设备树二进制文件（`.dtb`）在启动时传递给内核。内核包含一个设备树解析器。它会解析这个 `.dtb` 文件，从中找到名为 `memory` 的节点，该节点明确描述了系统物理内存的起始地址和大小。

2.探测法

探测法的核心原理是“暴力尝试读写”，即由操作系统主动向可疑的内存地址写入特定的测试数据（如`0x55AA`），并立即读回验证；若写入和读出的值一致，则认为该地址对应的物理内存单元存在且可用，否则便标记为保留或不存在。通过系统地遍历整个可能的地址空间来完成内存地图的绘制。这种方法简单直接，不依赖外部信息，但风险很高，因为向设备内存

（如显存）的误写可能导致系统崩溃。