南开大学操作系统实验报告

学号：2110598 姓名：许宸

学号：2113384 姓名：刘新宇

学号：2112487 姓名：刘轩宇

实验题目 — Lab 2 物理内存和页表

实验目的

- 理解页表的建立和使用方法
- 理解物理内存的管理方法
- 理解页面分配算法

环境

软件环境

ubuntu22.04，QEMU-4.1.1

实验步骤与内容

（一）理解 first-fit 连续物理内存分配算法

first-fit 连续物理内存分配算法是基础的物理内存分配方法。阅读实验手册的教程并结合kern/mm/default_pmm.c中的相关代码，分析default_init，default_init_memmap，default_alloc_pages，default_free_pages等相关函数，并描述程序在进行物理内存分配的过程以及各个函数的作用。 请在实验报告中简要说明你的设计实现过程。请回答如下问题：

default_init

初始化全局变量nr_free，记录可用的物理页面数，并初始化free_list，记录可用的物理页面的链表。

default_init_memmap

用于初始化一个新的空闲内存块，并将其添加到空闲列表中。
首先初始化n个物理内存页面，将它们的flag、property、ref设置为0，然后将第一个内存页的属性设置为n，表示块中有n个页可用，然后将内存块的属性设置为PageProperty，更新nr_free。
然后将初始化后的内存块插入到free_list中，其中判断了free_list是否为空，若为空，则base是第一个空闲内存块，将base直接添加到free_list，若free_list不为空，则遍历free_list，其中使用le2page将链表节点转换为Page结构，以获取当前遍历的页表地址信息，将base与当前遍历的页表地址比较，若base的地址小于page，则base应插入到page之前，并跳出循环，若base的地址大于page，则继续循环，若遍历到末尾也没有找到合适的位置，则直接插入到链表的末尾。

default_alloc_pages

用于分配连续的物理页面，返回分配页面的首地址。
首先检查是否有足够多的可用页满足分配请求，若没有，则返回NULL。
然后遍历free_list，寻找第一个足够多页数的空闲内存块，若找到了一个满足条件的内存块，则将其从free_list中删除，若内存块页数大于请求页数，将剩余的页组合成一个新的可用内存空，将其可用页数减去n，将新的内存块添加到free_list中，更新nr_free，最后返回请求内存块的指针。

default_free_pages

实现了内存释放，将一组连续的内存页释放会free_list中。
首先遍历从base开始的连续n个内存页，在每次循环中，使用assert检查内存页p的状态，确保既不是PageReserved也不是PageProperty，否则触发断言错误，将内存页的flag设置为0，表示这些页不再被分配或保留，将ref设置为0，表示目前没有任何引用指向该页。
将base内存块的property设置为n，表示内存块中有n个连续的内存页1可被分配，同时使用SetPageProperty函数将属性标记设置为PageProperty，表示这个内存块是一个空闲内存块。
将释放的内存块base添加会free_list中，若free_list为空，则直接添加到free_list，否则与default_init_memmap函数类似，将base内存块按照内存地址添加到对应位置。
接下来，尝试合并base内存块的前一个和后一个内存块，以减少内存碎片，首先，查找前一个内存块，如果前一个内存块存在且与base相邻，则可以合并，同理下一个内存块，当条件符合后执行合并操作，包括更新合并后的页面信息和地址。

first fit算法是否有进一步的改进空间?

1. 可能会存在大量的内存碎片，存在于已分配和未分配内存块之间的未使用的内存无法被有效利用。
2. 需要浏览整个内存空间的空闲块列表，导致分配效率较低。
3. 容易在空闲块列表中留下很小的空闲块，这些块难以利用，容易导致外部碎片。
4. 在多线程环境中，First Fit 算法需要加锁来保护空闲块列表，以防止多个线程同时进行分配和释放操作，导致性能问题。

（二）实现 Best-Fit 连续物理内存分配算法

参考kern/mm/default_pmm.c对First Fit算法的实现，编程实现Best Fit页面分配算法。

best_fit_init_memmap

初始化一组连续内存页与first fit算法类似。

```
best_fit_init_memmap(struct Page *base, size_t n) {
    assert(n > 0);
    struct Page *p = base;
    for (; p != base + n; p ++) {
        assert(PageReserved(p));

        /*LAB2 EXERCISE 2: YOUR CODE*/
        // 清空当前页框的标志和属性信息，并将页框的引用计数设置为0
        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);
             /*LAB2 EXERCISE 2: YOUR CODE*/
            // 编写代码
            // 1、当base < page时，找到第一个大于base的页，将base插入到它前面，并退出
循环
            // 2、当list_next(le) == &free_list时，若已经到达链表结尾，将base插入到链
表尾部
            if(base < page){
                list_add_before(le,&(base->page_link));
            }
            else if(list_next(le) == &free_list){
                list_add(le, &(base->page_link));
            }
        }
    }
}
```

best_fit_alloc_pages

在遍历可用页时使用best_fit变量记录满足需求且具有最小超出空间的内存块，以及smallest_gap变量记录最小的超出空间大小。初始化时，smallest_gap 被设置为size_t类型的最大值，以确保后续的比较可以找到更小的超出空间。
对于每个空闲内存块，函数检查是否满足需求（p->property >= n）且是否具有比之前找到的最小超出空间更小的超出空间。如果是，则更新 best_fit和smallest_gap。
其他代码与defaut_fit_alloc_pages函数类似。

```
// 下面的代码是best-fit的部分代码
// 遍历空闲链表，查找满足需求的最小空闲页框
list_entry_t *best_fit = NULL; // 记录最佳匹配
size_t smallest_gap = (size_t) -1; // 记录最小的超出的空闲空间，初始化为最大的size_t
值

while ((le = list_next(le)) != &free_list) {
   struct Page *p = le2page(le, page_link);
   // 检查是否满足需求并且空闲空间小于之前找到的最小空闲空间
   if (p->property >= n && (p->property - n) < smallest_gap) {
      best_fit = le;
      smallest_gap = p->property - n;
   }
}

// 如果找到了合适的页框
if (best_fit != NULL) {
   struct Page *page = le2page(best_fit, page_link);
```

```
    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;
}
```

best_fit_free_pages

代码与defaut_fit_alloc_pages函数类似。

```
static void
best_fit_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);
    }
    /*LAB2 EXERCISE 2: YOUR CODE*/
    // 编写代码
    // 具体来说就是设置当前页块的属性为释放的页块数、并将当前页块标记为已分配状态、最后增
加nr_free的值
    base->property = n;              // 设置页的属性为n
    SetPageProperty(base);          // 将页面设置为属性页
    nr_free += n;                   // 更新空闲页数
    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);
```

```
        /*LAB2 EXERCISE 2: YOUR CODE*/
         // 编写代码
        // 1、判断前面的空闲页块是否与当前页块是连续的，如果是连续的，则将当前页块合并到
前面的空闲页块中
        // 2、首先更新前一个空闲页块的大小，加上当前页块的大小
        // 3、清除当前页块的属性标记，表示不再是空闲页块
        // 4、从链表中删除当前页块
        // 5、将指针指向前一个空闲页块，以便继续检查合并后的连续空闲页块'
        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));
        }
    }
}
```

```
>>>>>>>>>> here_make>>>>>>>>>>>
<<<<<<<<<<<<<< here_run_qemu <<<<<<<<<<<<<<<<<<
try to run qemu
qemu pid=2893
<<<<<<<<<<<<<< here_run_check <<<<<<<<<<<<<<<<<<
  -check physical_memory_map_information:    OK
  -check_best_fit:                           OK
  -check ticks:                              OK
Total Score: 30/30
```

Best-Fit 算法是否有进一步的改进空间?

1. 仍然无法完全消除内部碎片和外部碎片。
2. 仍需要遍历整个空闲内存块链表以找到最适合大小的内存块。

改进空间:

1. 实现缓存机制，以记录先前找到的最佳匹配，从而加速后续的分配操作。
2. 周期性地尝试合并相邻的小内存块，以减少外部碎片。

（三）Challenge：buddy system（伙伴系统）分配算法

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... 以下是算法描述：

分配内存：

1. 寻找大小合适的内存块（大于等于所需大小并且最接近2的幂）
    1. 如果找到了，分配给应用程序。
    2. 如果没找到，分出合适的内存块。
        1. 对半分离出高于所需大小的空闲内存块
        2. 如果分到最低限度，分配这个大小。
        3. 回溯到步骤1（寻找合适大小的块）
        4. 重复该步骤直到一个合适的块

释放内存：

1. 释放该内存块
2. 寻找相邻的块，看其是否释放了。
3. 如果相邻块也释放了，合并这两个块，重复上述步骤直到遇上未释放的相邻块，或者达到最高上限（即所有内存都释放了）。

源代码：

```
#include <pmm.h>
#include <buddy_system.h>

struct buddy {
    size_t size;
    uintptr_t *longest;
    size_t longest_num;
    size_t total_num;
    size_t curr_free;
    struct Page *begin_page;
};

struct buddy b[MAX_BUDDY_NUMBER];
int id_ = 0;

static size_t next_power_of_2(size_t size) {
    size |= size >> 1;
    size |= size >> 2;
    size |= size >> 4;
    size |= size >> 8;
    size |= size >> 16;
    return size + 1;
}

static void
buddy_init() {

}
```

```
static void
buddy_init_memmap(struct Page *base, size_t n) {
    cprintf("n: %d\n", n);
    struct buddy *buddy = &b[id_++];

    size_t s = next_power_of_2(n);  // 分配大于等于所需大小并且最接近2的幂
    size_t extra = s - n;
    size_t e = next_power_of_2(extra);

    buddy->size = s;
    buddy->curr_free = s - e;
    buddy->longest = KADDR(page2pa(base));  // 将物理地址转换为虚拟地址
    buddy->begin_page = pa2page(PADDR(ROUNDUP(buddy->longest + 2 * s *
sizeof(uintptr_t), PGSIZE)));    // 将虚拟地址转换为物理地址
    buddy->longest_num = buddy->begin_page - base;
    buddy->total_num = n - buddy->longest_num;

    size_t sn = buddy->size * 2;    // 2 * buddy->size - 1为最大的叶子节点

    for (int i = 0; i < 2 * buddy->size - 1; i++) {
        if (IS_POWER_OF_2(i + 1)) {
            sn /= 2;
        }
        buddy->longest[i] = sn;
    }

    int id = 0;
    while (1) {
        if (buddy->longest[id] == e) {
            buddy->longest[id] = 0;
            break;
        }
        id = RIGHT_LEAF(id);
    }
    // 更新父节点的值
    while (id) {
        id = PARENT(id);
        buddy->longest[id] = MAX(buddy->longest[LEFT_LEAF(id)], buddy-
>longest[RIGHT_LEAF(id)]);
    }
    // 初始化
    struct Page *p = buddy->begin_page;
    for (; p != base + buddy->curr_free; p ++) {
        assert(PageReserved(p));
        p->flags = p->property = 0;
        set_page_ref(p, 0);
    }
}

static struct Page *
buddy_alloc_pages(size_t n) {
    assert(n > 0);
    if (!IS_POWER_OF_2(n))
```

```
        n = next_power_of_2(n);

    size_t id = 0;
    size_t sn;
    size_t offset = 0;

    struct buddy *buddy = NULL;
    for (int i = 0; i < id_; i++) {
        if (b[i].longest[id] >= n) {
            buddy = &b[i];
            break;
        }
    }

    if (!buddy) {
        return NULL;
    }

    for (sn = buddy->size; sn != n; sn /= 2) {
        if (buddy->longest[LEFT_LEAF(id)] >= n)
            id = LEFT_LEAF(id);
        else
            id = RIGHT_LEAF(id);
    }

    buddy->longest[id] = 0;
    offset = (id + 1) * sn - buddy->size;
    // 更新父节点的值
    while (id) {
        id = PARENT(id);
        buddy->longest[id] = MAX(buddy->longest[LEFT_LEAF(id)], buddy-
>longest[RIGHT_LEAF(id)]);
    }

    buddy->curr_free -= n;

    return buddy->begin_page + offset;
}


static void
buddy_free_pages(struct Page *base, size_t n) {
    struct buddy *buddy = NULL;

    for (int i = 0; i < id_; i++) {
        struct buddy *t = &b[i];
        if (base >= t->begin_page && base < t->begin_page + t->size) {
            buddy = t;
        }
    }

    if (!buddy) return;

    unsigned sn, id = 0;
```

```
    unsigned left_longest, right_longest;
    unsigned offset = base - buddy->begin_page; // 页的偏移量

    assert(offset >= 0 && offset < buddy->size);

    sn = 1;
    id = offset + buddy->size - 1;

    for (; buddy->longest[id]; id = PARENT(id)) {
        sn *= 2;
        if (id == 0)
            return;
    }

    buddy->longest[id] = sn;
    buddy->curr_free += sn;

    while (id) {
        id = PARENT(id);
        sn *= 2;

        left_longest = buddy->longest[LEFT_LEAF(id)];
        right_longest = buddy->longest[RIGHT_LEAF(id)];

        if (left_longest + right_longest == sn)
            buddy->longest[id] = sn;
        else
            buddy->longest[id] = MAX(left_longest, right_longest);
    }

}


static size_t
buddy_nr_free_pages(void) {
    size_t total_free_pages = 0;
    for (int i = 0; i < id_; i++) {
        total_free_pages += b[i].curr_free;
    }
    return total_free_pages;
}


static void
buddy_check(void) {

    cprintf("New test case: testing memory block validation...\n");

    // 分配一页内存
    struct Page *p_ = buddy_alloc_pages(1);  // 假定1表示一页
    assert(p_ != NULL);

    // 获取页面的物理地址，并转换为可用的虚拟地址。这里需要根据你的实现来完成。
    // 注意：你可能需要使用其他函数来获取/转换地址，依据你的内核/平台实现。
```

```
    uintptr_t pa = page2pa(p_);
    uintptr_t *va = KADDR(pa);

    // 写入数据到分配的内存块
    int *data_ptr = (int *)va;
    *data_ptr = 0xdeadbeef;  // 写入一个魔数，稍后用于验证

    // 读取并验证数据
    assert(*data_ptr == 0xdeadbeef);

    // 释放内存块
    buddy_free_pages(p_, 1);

    // 验证是否可以正常释放，例如再次分配相同的内存块并检查地址是否相同
    struct Page *p_2 = buddy_alloc_pages(1);
    assert(p_ == p_2);  // 假定相同的内存块地址会被重新分配，这取决于你的内存分配器实现

    // 清理
    buddy_free_pages(p_2, 1);

    cprintf("Memory block validation test passed!\n");
}



const struct pmm_manager buddy_pmm_manager = {
        .name = "buddy_pmm_manager",
        .init = buddy_init,
        .init_memmap = buddy_init_memmap,
        .alloc_pages = buddy_alloc_pages,
        .free_pages = buddy_free_pages,
        .nr_free_pages = buddy_nr_free_pages,
        .check = buddy_check,
};
```

（四）任意大小的内存单元slub分配算法

源代码

```
#include <slub.h>
#include <list.h>
#include <defs.h>
#include <string.h>
#include <stdio.h>
#define CACHE_NAMELEN 16

// 定义内存缓存结构
struct kmem_cache_t {
    list_entry_t slabs_full;        // 完全分配的内存块列表
    list_entry_t slabs_partial;     // 部分分配的内存块列表
    list_entry_t slabs_free;        // 空闲的内存块列表
```

```
    uint16_t objsize;                // 每个对象的大小
    uint16_t num;                    // 每个内存块可以容纳的对象数量
    void (*ctor)(void*, struct kmem_cache_t *, size_t); // 构造函数指针
    void (*dtor)(void*, struct kmem_cache_t *, size_t); // 析构函数指针
    char name[CACHE_NAMELEN];        // 缓存名称
    list_entry_t cache_link;         // 缓存链表节点
};

// 内存块结构
struct slab_t {
    int ref;                         // 引用计数
    struct kmem_cache_t *cachep;     // 所属缓存
    uint16_t inuse;                  // 已使用对象数量
    uint16_t free;                   // 空闲对象数量
    list_entry_t slab_link;          // 内存块链表节点
};

#define SIZED_CACHE_NUM 8
#define SIZED_CACHE_MIN 16
#define SIZED_CACHE_MAX 2048

#define le2slab(le, link) ((struct slab_t*)le2page((struct Page*)le, link))
#define slab2kva(slab) (page2kva((struct Page*)slab))

static list_entry_t cache_chain;         // 缓存链表头
static struct kmem_cache_t cache_cache; // 用于管理缓存的缓存
static struct kmem_cache_t *sized_caches[SIZED_CACHE_NUM]; // 各个大小的内存缓存数组

// 内存缓存创建函数
struct kmem_cache_t *kmem_cache_create(const char *name, size_t size,
                                       void (*ctor)(void*, struct kmem_cache_t *,
size_t),
                                       void (*dtor)(void*, struct kmem_cache_t *,
size_t)) {
    assert(size <= (PGSIZE - 2)); // 确保对象大小不超过一页大小
    struct kmem_cache_t *cachep = kmem_cache_alloc(&(cache_cache)); // 分配内存缓存
结构
    if (cachep != NULL) {
        cachep->objsize = size; // 设置对象大小
        cachep->num = PGSIZE / (sizeof(int16_t) + size); // 计算每个内存块可以容纳的
对象数量
        cachep->ctor = ctor; // 设置构造函数指针
        cachep->dtor = dtor; // 设置析构函数指针
        memcpy(cachep->name, name, CACHE_NAMELEN); // 复制缓存名称
        list_init(&(cachep->slabs_full)); // 初始化完全分配的内存块列表
        list_init(&(cachep->slabs_partial)); // 初始化部分分配的内存块列表
        list_init(&(cachep->slabs_free)); // 初始化空闲的内存块列表
        list_add(&(cache_chain), &(cachep->cache_link)); // 将缓存添加到缓存链表中
    }
    return cachep;
}

// 内存缓存销毁函数
void kmem_cache_destroy(struct kmem_cache_t *cachep) {
```

```
    list_entry_t *head, *le;

    // 销毁完全分配的内存块
    head = &(cachep->slabs_full);
    le = list_next(head);
    while (le != head) {
        list_entry_t *temp = le;
        le = list_next(le);
        kmem_slab_destroy(cachep, le2slab(temp, page_link));
    }

    // 销毁部分分配的内存块
    head = &(cachep->slabs_partial);
    le = list_next(head);
    while (le != head) {
        list_entry_t *temp = le;
        le = list_next(le);
        kmem_slab_destroy(cachep, le2slab(temp, page_link));
    }

    // 销毁空闲的内存块
    head = &(cachep->slabs_free);
    le = list_next(head);
    while (le != head) {
        list_entry_t *temp = le;
        le = list_next(le);
        kmem_slab_destroy(cachep, le2slab(temp, page_link));
    }

    // 释放内存缓存结构
    kmem_cache_free(&(cache_cache), cachep);
}

// 分配一个对象
void *kmem_cache_alloc(struct kmem_cache_t *cachep) {
    list_entry_t *le = NULL;

    // 在部分分配的内存块列表中查找可用内存块
    if (!list_empty(&(cachep->slabs_partial)))
        le = list_next(&(cachep->slabs_partial));
    // 如果部分分配的内存块列表为空，则尝试从空闲内存块列表中获取内存块，如果失败则尝试增
长内存块
    else {
        if (list_empty(&(cachep->slabs_free)) && kmem_cache_grow(cachep) == NULL)
            return NULL;
        le = list_next(&(cachep->slabs_free));
    }

    list_del(le);
    struct slab_t *slab = le2slab(le, page_link);
    void *kva = slab2kva(slab);
    int16_t *bufctl = kva;
    void *buf = bufctl + cachep->num;
    void *objp = buf + slab->free * cachep->objsize;
```

```
    slab->inuse++;
    slab->free = bufctl[slab->free];
    if (slab->inuse == cachep->num)
        list_add(&(cachep->slabs_full), le);
    else
        list_add(&(cachep->slabs_partial), le);
    return objp;
}

// 分配一个对象并清零
void *kmem_cache_zalloc(struct kmem_cache_t *cachep) {
    void *objp = kmem_cache_alloc(cachep);
    memset(objp, 0, cachep->objsize);
    return objp;
}

// 释放一个对象
void kmem_cache_free(struct kmem_cache_t *cachep, void *objp) {
    void *base = page2kva(pages);
    void *kva = ROUNDDOWN(objp, PGSIZE);
    struct slab_t *slab = (struct slab_t *)&pages[(kva - base) / PGSIZE];
    int16_t *bufctl = kva;
    void *buf = bufctl + cachep->num;
    int offset = (objp - buf) / cachep->objsize;

    list_del(&(slab->slab_link));
    bufctl[offset] = slab->free;
    slab->inuse--;
    slab->free = offset;
    if (slab->inuse == 0)
        list_add(&(cachep->slabs_free), &(slab->slab_link));
    else
        list_add(&(cachep->slabs_partial), &(slab->slab_link));
}

// 获取内存对象的大小
size_t kmem_cache_size(struct kmem_cache_t *cachep) {
    return cachep->objsize;
}

// 获取内存缓存的名称
const char *kmem_cache_name(struct kmem_cache_t *cachep) {
    return cachep->name;
}

// 销毁空闲内存块列表中的所有内存块
int kmem_cache_shrink(struct kmem_cache_t *cachep) {
    int count = 0;
    list_entry_t *le = list_next(&(cachep->slabs_free));
    while (le != &(cachep->slabs_free)) {
        list_entry_t *temp = le;
        le = list_next(le);
        kmem_slab_destroy(cachep, le2slab(temp, page_link));
```

```
            count++;
        }
        return count;
    }

    // 销毁所有空闲内存块列表中的内存块
    int kmem_cache_reap() {
        int count = 0;
        list_entry_t *le = &(cache_chain);
        while ((le = list_next(le)) != &(cache_chain))
            count += kmem_cache_shrink(to_struct(le, struct kmem_cache_t,
    cache_link));
        return count;
    }

    // 分配指定大小的内存
    void *kmalloc(size_t size) {
        assert(size <= SIZED_CACHE_MAX);
        return kmem_cache_alloc(sized_caches[kmem_sized_index(size)]);
    }

    // 释放内存
    void kfree(void *objp) {
        void *base = slab2kva(pages);
        void *kva = ROUNDDOWN(objp, PGSIZE);
        struct slab_t *slab = (struct slab_t *)&pages[(kva - base) / PGSIZE];
        kmem_cache_free(slab->cachep, objp);
    }

    void kmem_int() {
        cache_cache.objsize = sizeof(struct kmem_cache_t);
        cache_cache.num = PGSIZE / (sizeof(int16_t) + sizeof(struct kmem_cache_t));
        cache_cache.ctor = NULL;
        cache_cache.dtor = NULL;
        memcpy(cache_cache.name, cache_cache_name, CACHE_NAMELEN);
        list_init(&(cache_cache.slabs_full));
        list_init(&(cache_cache.slabs_partial));
        list_init(&(cache_cache.slabs_free));
        list_init(&(cache_chain));
        list_add(&(cache_chain), &(cache_cache.cache_link));

        for (int i = 0, size = 16; i < SIZED_CACHE_NUM; i++, size *= 2)
            sized_caches[i] = kmem_cache_create(sized_cache_name, size, NULL, NULL);
    }
```

（五）Challenge：硬件的可用物理内存范围的获取方法

1. BIOS 调用：一种方式是通过 BIOS 调用来获取可用的物理内存范围。BIOS 在启动时负责初始化硬件，包括内存控制器，因此它通常具有关于物理内存的信息。操作系统可以通过访问 BIOS 中的数据结构或使用 BIOS 调用来获取内存信息。
2. UEFI：对于使用 UEFI 启动的计算机，可以通过 UEFI 固件接口来获取系统信息，包括可用的物理内存范围。UEFI 提供了一组协议，如EFI_BOOT_SERVICES，可以用于查询系统信息，包括物理内存映射。

3. 内存映射表：在一些体系结构上，特别是在一些嵌入式系统上，内存映射表会描述可用的物理内存范围。操作系统可以查阅这些表来获取内存信息。
4. 物理内存探测：在某些情况下，操作系统可以进行物理内存的探测，例如从某个地址开始，逐渐探测可用内存范围，直到遇到不可用内存。这种方式相对复杂，因为需要考虑硬件的特定性。
5. 操作系统支持：一些现代操作系统已经具备了自动检测和管理物理内存的功能，可以自动获取并管理可用的物理内存范围。