OS Lab2实验报告

练习1：理解first-fit 连续物理内存分配算法（思考题）

1. `default_init`

该函数初始化空闲列表和空闲页面计数器`nr_free`。在初始化时，调用`list_init`来设置空闲列表为空，并将空闲页面计数器设为0。

2. `default_init_memmap`

这个函数负责初始化一个空闲块。它接收两个参数：起始地址`base`和页面数量`n`。该函数将`n`个页面标记为有效空闲，设置属性，并将它们添加到空闲列表中。如果当前空闲列表不为空，则按照地址顺序插入新的空闲块。

3. `default_alloc_pages`

这个函数用于分配指定数量的连续页面。它会遍历空闲列表，找到第一个满足请求的空闲块。如果找到合适的块，它会更新相应的属性，并从空闲列表中删除该块，同时返回分配的页面的地址。如果没有找到合适的块，则返回`NULL`。

4. `default_free_pages`

该函数负责释放指定数量的连续页面并将其重新添加到空闲列表中。它会首先检查释放的页面是否可以与相邻的空闲块合并，从而减少内存碎片。合并后，更新页面属性，并在合适的位置插入释放的块。

设计改进空间

1. **内存碎片管理**：
    - first-fit算法容易导致内存碎片的产生。可以考虑在分配时使用更先进的策略，例如最佳适应（Best-Fit）或合并相邻的空闲块来减少碎片。
2. **合并逻辑优化**：
    - 在释放页面时，可以选择延迟合并策略，仅在必要时合并空闲块，以避免频繁的内存操作。并且优化合并逻辑，确保能够在所有可能的情况下都能合并相邻的空闲块，减少内存碎片。
3. **动态调整空闲列表**：
    - 当系统负载变化时，可以动态调整空闲列表的管理策略，以提高内存分配的效率。
4. **使用更复杂的数据结构**：
    - 考虑使用更复杂的数据结构（如红黑树或线段树）来管理空闲块，以加快搜索速度。

练习2：实现 Best-Fit 连续物理内存分配算法（需要编程）

设计与实现过程

在实现Best Fit算法时，主要步骤包括：

- best_fit_init(void) 初始化：
  设置空闲列表和空闲块的数量。
- best_fit_init_memmap 内存映射初始化：
  在创建新内存块时，设置块的属性，并将其插入空闲列表。
- best_fit_alloc_pages(size_t n) 内存分配：
  遍历空闲列表，找到最小的满足请求的块并进行分配。如果找到的块比请求的块大，则更新剩余的空闲块的属性。

```
// 核心代码：如果找到满足需求的页面，记录该页面以及当前找到的最小连续空闲页框数量
    while ((le = list_next(le)) != &free_list) {
        struct Page *p = le2page(le, page_link);
        if (p->property >= n) {
            if(min_size > p->property)
            {
                page=p;
                min_size=p->property;
            }
        }
    }
```

- best_fit_free_pages(struct Page *base, size_t n) 内存释放：
  在释放内存时，将释放的块插入空闲列表，并尝试合并相邻的空闲块。

Best Fit与First Fit的关键区别

1. 分配策略

   First Fit: 从空闲列表的头部开始搜索，找到第一个满足请求大小的空闲块并分配。如果该块比请求的大小大，通常会将其分割，剩余部分仍然放入空闲列表。 Best Fit: 遍历整个空闲列表，寻找最小的能够满足请求的空闲块。这种方法确保分配的块是最小的，尽量减少内存碎片。

2. 时间复杂度

   First Fit: 在最坏情况下，时间复杂度为O(n)，因为可能需要遍历整个列表，找到第一个合适的块。 Best Fit: 同样在最坏情况下，时间复杂度为O(n)，但由于需要比较每个块的大小，通常在性能上可能比First Fit稍慢。

3. 内存利用率

   First Fit: 可能会留下较大的空闲块，导致内存碎片较多。 Best Fit: 通过选择最小的适合块，能更好地利用内存，减少碎片，但可能导致较多的小空闲块。

进一步改进空间

Best Fit算法的改进空间包括：

- 合并空闲块: 在释放内存时，优化合并相邻空闲块的策略，以减少内存碎片。
- 维护链表: 考虑使用更高效的数据结构（如平衡树）来维护空闲块，以提高搜索效率。
- 优化搜索策略: 可以引入记忆化技术，记录常用的块的大小，以加速后续分配请求的处理。

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...

寻找内存地址

1.**判断要分配的内存合规**

- 如果请求的内存大小 `size` 不是2的幂，通过 `fixsize` 函数调整为大于等于 `size` 的最小的2的幂。
- 检查根节点的最长空闲块是否小于请求的大小

2.**查找合适的空闲块**:

- 通过遍历二叉树结构，找到第一个足够大的空闲块。二叉树的每个节点代表一个内存块，左子节点代表较小的一半，右子节点代表较大的一半。
- 从根节点开始，比较当前节点的大小与请求的大小。如果当前节点的左子节点有足够大的空间，则移动到左子节点；如果右子节点有足够大的空间或左右子节点大小相等，则移动到右子节点。

3.**标记空闲块**:

- 一旦找到合适的空闲块，将其标记为已使用（`self[index].longest = 0`）
- 计算找到的空闲块的偏移量
- 回溯到根节点，更新每个祖先节点的 `longest` 属性，以记录子节点中最大的空闲块大小

```
`int buddy2_alloc(struct buddy2* self, int size) {`
  `unsigned index = 0;`
  `unsigned node_size;`
  `unsigned offset = 0;`

  `for (node_size = self->size; node_size != size; node_size /= 2) {`
    `if (self[LEFT_LEAF(index)].longest >= size) {`
      `if (self[RIGHT_LEAF(index)].longest >= size) {`
        `index = (self[LEFT_LEAF(index)].longest <=
self[RIGHT_LEAF(index)].longest) ?`
                `LEFT_LEAF(index) : RIGHT_LEAF(index);`
      `} else {`
        `index = LEFT_LEAF(index);`
      `}`
    `} else {`
      `index = RIGHT_LEAF(index);`
    `}`
  `}`
```

```
  `self[index].longest = 0;`
  `offset = (index + 1) * node_size - self->size;`
  `while (index) {`
    `index = PARENT(index);`
    `self[index].longest = MAX(self[LEFT_LEAF(index)].longest,
self[RIGHT_LEAF(index)].longest);`
  `}`
  ``return offset;`
`}`
```

分配内存

在这个函数中我们会调用buddy2_alloc(root, n)函数获取要分配的块的偏移量，使用循环的方式在链表中找到对应块，修改该块的属性，记录分配信息，进一步更新空闲页面数。

```
static struct Page*`
`buddy_alloc_pages(size_t n) {`
  `rec[nr_block].offset = buddy2_alloc(root, n);`
  `int i;`
  `for (i = 0; i < rec[nr_block].offset + 1; i++)`
    `le = list_next(le);`
  `page = le2page(le, page_link);`
  `int allocpages;`
  `if (!IS_POWER_OF_2(n))`
    `allocpages = fixsize(n);`
  `else {`
     `allocpages = n;`
  `}`
  `for (i = 0; i < allocpages; i++) {`
    `temp_le = list_next(le);`
    `p = le2page(le, page_link);`
   `ClearPageProperty(p);`
  `le = temp_le;`
```

```
    `}`
`}
```

释放内存

1.**恢复页面**

- 更新空闲页面数`nr_free += allocpages;` 将释放的页面数加到总空闲页面数中。
- 恢复页面属性并标记为空闲

```
`nr_free += allocpages;`
`struct Page* p;`
`self[index].longest = allocpages;`
`for (i = 0; i < allocpages; i++) {`
 `p = le2page(le, page_link);`
 `p->flags = 0;`
 `p->property = 1;`
 `SetPageProperty(p);`
 `le = list_next(le);`
`}
```

2.**合并空闲块**

检查是否可以合并:

- `if (left_longest + right_longest == node_size)`：检查左子节点和右子节点的空闲长度之和是否等于当前节点的总大小。如果是，这意味着当前节点的两个子节点都是空闲的，并且它们的总大小正好等于父节点的大小，因此可以合并。

合并操作:

- `self[index].longest = node_size;`：如果两个子节点可以合并，将父节点的空闲长度设置为`node_size`，表示现在父节点是一个更大的空闲块。

不合并的情况:

- `else self[index].longest = MAX(left_longest, right_longest);`：如果两个子节点不能合并（即它们不都是空闲的，或者它们的总大小不等于父节点的大小），则将父节点的空闲长度设置为左子节点和右子节点中较长的一个。

```
 while (index) {
  index = PARENT(index);
  node_size *= 2;
  left_longest = self[LEFT_LEAF(index)].longest;
  right_longest = self[RIGHT_LEAF(index)].longest;
  if (left_longest + right_longest == node_size)
   self[index].longest = node_size;
  else
   self[index].longest = MAX(left_longest, right_longest);
 }`
```

检查函数

按照网站上提供样例的分配方式依次分配内存再释放，通过输出内存块的地址判断分配是否正确，通过断言判断内存块释放后是否正确合并。

```
buddy_check(void) {
    struct Page *p0, *A, *B,*C,*D;
    A=alloc_pages(70);
    B=alloc_pages(35);
    assert(A+128==B);
    cprintf("A %p\n",A);
    cprintf("B %p\n",B);
    C=alloc_pages(80);
    assert(A+256==C);
    cprintf("C %p\n",C);
    free_pages(A,70);
    cprintf("B %p\n",B);
    D=alloc_pages(60);
    cprintf("D %p\n",D);
    assert(B+64==D);
    free_pages(B,35);
    cprintf("D %p\n",D);
    free_pages(D,60);
    cprintf("C %p\n",C);
    free_pages(C,80);
}
```

输出结果如下：

```
A 0xffffffffc04864c0
B 0xffffffffc04878c0
C 0xffffffffc0488cc0
B 0xffffffffc04878c0
D 0xffffffffc04882c0
D 0xffffffffc04882c0
C 0xffffffffc0488cc0
```

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。

整体思路

slub算法是linux内核中用于分配内存单元的算法，对于整页内存的需求，可以直接使用buddy system算法获得；对于较小内存块的需求，使用slub算法将整页内存分为较小块的内存，实现零碎内存的分配和回收。

在我的实现过程中，我把内存分组管理，每个组分别包含2^3、2^4、...2^11个字节，在4K页大小的默认情况下，另外还有两个特殊的组，分别是96B和192B，共11组。

结构定义

首先，需要定义页的最大缓存大小、缓存数组以及一个页的大小，便于接下来计算每个slab内可以分配的数量

```
#define PAGE_SIZE 4096
#define MAX_CACHE_SIZE 2048 // 最大缓存大小
#define CACHE_COUNT 12 // 缓存组数量
#define SLAB_MAX_COUNT 10
```

然后，定义了几个**结构体**，整体使用单向链表来实现next的查询。

- kmem_cache结构

```
// kmem_cache结构
typedef struct KmemCache {
    Slab *partial;      // 部分使用的slab
    Slab *full;         // 完全使用的slab
    int object_size;    // 每个对象的大小
    Object objects[SLAB_MAX_COUNT]; // 静态数组用于对象
} KmemCache;
```

- slab结构

```
typedef struct Slab {
    Object *free_list;  // 空闲对象链表
    int in_use;         // 当前使用的对象数量
    struct Slab *next;  // 指向下一个slab
} Slab;
```

- 对象结构

```
typedef struct Object {
    struct Object *next;  // 指向下一个对象
} Object;
```

结构体定义之后，我使用`KmemCache kmalloc_caches[CACHE_COUNT];`来实现分配12种内存情况。

除此以外，我还定义了内存池来避免动态分配（即不使用malloc和free函数）

```
// 内存池大小
#define POOL_SIZE (CACHE_COUNT * SLAB_MAX_COUNT * sizeof(Object))

// 定义内存池
static char memory_pool[POOL_SIZE];
static int pool_index = 0;

// 从内存池中分配内存
void *pool_alloc(size_t size) {
    if (pool_index + size > POOL_SIZE) {
        return NULL; // 内存池不足
    }
    void *ptr = memory_pool + pool_index;
    pool_index += size;
    return ptr;
}

// 释放内存池中的内存（这里只是简单的重置，实际使用中需要管理）
void pool_free() {
    pool_index = 0; // 简单重置内存池
}
```

主要函数

- **初始化kmem_cache**，因为定义了大小为12的一维KmemCache数组，因此需要实现每种内存大小的KmemCache的初始化。在初始化时，我首先将对象链表初始化为空，然后使用内存池来分配对象，并链接到对象链表中。

```
void kmem_cache_init() {
    for (int i = 0; i < CACHE_COUNT; i++) {
```

```
        if (i < 10) {
            kmalloc_caches[i].object_size = 1 << (i + 3); // 2^3到2^11
        } else if (i == 10) {
            kmalloc_caches[i].object_size = 96; // 96字节
        } else {
            kmalloc_caches[i].object_size = 192; // 192字节
        }
        kmalloc_caches[i].partial = NULL;
        kmalloc_caches[i].full = NULL;

        // 初始化对象链表
        kmalloc_caches[i].partial = pool_alloc(sizeof(Slab)); // 从内存池分配slab
        assert(kmalloc_caches[i].partial != NULL);
        kmalloc_caches[i].partial->free_list = NULL;
        kmalloc_caches[i].partial->in_use = 0;

        for (int j = 0; j < SLAB_MAX_COUNT; j++) {
            Object *obj = &kmalloc_caches[i].objects[j]; // 使用静态数组
            obj->next = kmalloc_caches[i].partial->free_list;
            kmalloc_caches[i].partial->free_list = obj;
        }
    }
    cprintf("kmalloc caches initialized.\n");
}
```

- **内存分配slub_alloc**，在分配内存时，我首先判断是否需要创建新的slab，如果当前slab已经用完，则创建新的slab，并将其链接到对应的partial链表中。然后从slab的free_list中取出一个对象，并将其返回。需要注意的是，当从partial分配内存后，该slab的空间全被占满，则将该slab移至full中。

```
void *slub_alloc(size_t size) {
    for (int i = 0; i < CACHE_COUNT; i++) {
        if (kmalloc_caches[i].object_size == size) {
            KmemCache *cache = &kmalloc_caches[i];
            if (cache->partial) {
                Slab *slab = cache->partial;

                // 从空闲链表中取出一个对象
                if (slab->free_list) {
                    Object *obj = slab->free_list;
                    slab->free_list = obj->next;
                    slab->in_use++;

                    // 如果slab满了，移到full链表
                    if (slab->in_use == SLAB_MAX_COUNT) {
                        slab->next = cache->full; // 移到full链表
                        cache->full = slab;
                        cache->partial = NULL; // 清空partial链表
                    }
                    return obj; // 返回对象指针
                }
            }
```

```
            // 如果没有部分slab，则创建一个新的
            Slab *new_slab = (Slab *)pool_alloc(sizeof(Slab));
            assert(new_slab != NULL);
            new_slab->free_list = NULL;
            new_slab->in_use = 0;

            // 初始化对象链表
            for (int j = 0; j < SLAB_MAX_COUNT; j++) {
                Object *obj = &kmalloc_caches[i].objects[j];
                obj->next = new_slab->free_list;
                new_slab->free_list = obj;
            }

            new_slab->next = cache->partial;
            cache->partial = new_slab;

            // 返回第一个空闲对象
            return slub_alloc(size);
        }
    }

    return NULL; // 如果没有匹配的大小，则返回NULL
}
```

- **内存释放slub_free**，在释放内存时，找到与要被释放的空间大小一致的kmem_cache，然后找到对应的slab，并从slab的free_list中释放对象。若释放内存后，该slab中所有空间都没有被占用，则移除此slab

```
void slub_free(void *ptr, size_t size) {
    for (int i = 0; i < CACHE_COUNT; i++) {
        if (kmalloc_caches[i].object_size == size) {
            KmemCache *cache = &kmalloc_caches[i];
            Slab *slab = cache->partial;

            // 直接释放到该slab的空闲链表
            if (slab) {
                Object *obj = (Object *)ptr; // 将指针转换为 Object *
                obj->next = slab->free_list;
                slab->free_list = obj;
                slab->in_use--;

                // 如果slab变为空，移除该slab
                if (slab->in_use == 0) {
                    cache->partial = NULL; // 清空partial链表
                }
            }
            return;
        }
    }
}
```

- **检查函数slub_check**,首先打印每种大小的kmem_cache的初始状态，然后为每种大小的kmem_cache分配一个对象，然后释放，以此来检查内存分配和释放的正确性。

```
void slub_check() {
    cprintf("SLUB check begin\n");

    for (int i = 0; i < CACHE_COUNT; i++) {
        cprintf("Cache %d: size %d, partial slabs: %p, full slabs: %p\n",
                i, kmalloc_caches[i].object_size, kmalloc_caches[i].partial,
kmalloc_caches[i].full);
    }

    for (int i = 0; i < CACHE_COUNT; i++) {
        void *obj = slub_alloc(kmalloc_caches[i].object_size);
        assert(obj != NULL);
        cprintf("Allocated object of size %d: %p\n",
kmalloc_caches[i].object_size, obj);
        slub_free(obj, kmalloc_caches[i].object_size);
    }

    destroy_kmem_cache();
    cprintf("kmem_cache destroyed successfully\n");
    cprintf("SLUB check end\n");
}
```

运行结果：

```
SLUB check begin
Cache 0: size 8, partial slabs: 0xffffffffc02054f0, full slabs: 0x0
Cache 1: size 16, partial slabs: 0xffffffffc0205508, full slabs: 0x0
Cache 2: size 32, partial slabs: 0xffffffffc0205520, full slabs: 0x0
Cache 3: size 64, partial slabs: 0xffffffffc0205538, full slabs: 0x0
Cache 4: size 128, partial slabs: 0xffffffffc0205550, full slabs: 0x0
Cache 5: size 256, partial slabs: 0xffffffffc0205568, full slabs: 0x0
Cache 6: size 512, partial slabs: 0xffffffffc0205580, full slabs: 0x0
Cache 7: size 1024, partial slabs: 0xffffffffc0205598, full slabs: 0x0
Cache 8: size 2048, partial slabs: 0xffffffffc02055b0, full slabs: 0x0
Cache 9: size 4096, partial slabs: 0xffffffffc02055c8, full slabs: 0x0
Cache 10: size 96, partial slabs: 0xffffffffc02055e0, full slabs: 0x0
Cache 11: size 192, partial slabs: 0xffffffffc02055f8, full slabs: 0x0
Allocated object of size 8: 0xffffffffc0205070
Allocated object of size 16: 0xffffffffc02050d8
Allocated object of size 32: 0xffffffffc0205140
Allocated object of size 64: 0xffffffffc02051a8
Allocated object of size 128: 0xffffffffc0205210
Allocated object of size 256: 0xffffffffc0205278
Allocated object of size 512: 0xffffffffc02052e0
Allocated object of size 1024: 0xffffffffc0205348
Allocated object of size 2048: 0xffffffffc02053b0
Allocated object of size 4096: 0xffffffffc0205418
```

```
Allocated object of size 96: 0xffffffffc0205480
Allocated object of size 192: 0xffffffffc02054e8
kmem_cache destroyed successfully
```

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？

1. 分段检测物理内存占用情况

可以通过分段检测物理内存是否被占用，以获得未被占用的物理内存。这一方法的基本思路如下：

- **写入测试**: 向内存中的各个段写入特定数据（如模式数据或特定标识符）。
- **读取验证**: 随后从同一地址读取数据，判断是否能够成功读取。如果读取的数据与写入的数据一致，说明该段内存是可用的；如果读取的数据是零或其他预定义的值，则可以认为该段内存可能被占用。
- **边界检测**: 通过这种方式，可以逐步扩展已检测的可用内存范围，最终确定可用物理内存的边界。

2. 利用BIOS的内存检测功能

在系统启动时，操作系统可以调用BIOS提供的内存检测功能。这一方法的步骤如下：

- **调用BIOS中断**: 利用BIOS提供的中断（如`INT 15h`）来获取系统内存的布局信息。
- **获取内存信息**: BIOS会返回一个结构体，包含可用物理内存的起始地址和大小等信息。
- **结构体解析**: 操作系统可以解析该结构体，以确定可用的物理内存范围。这种方法通常可靠，因为BIOS在启动过程中会进行全面的内存检查。

3. 使用系统引导加载程序

引导加载程序在操作系统启动时执行，它可以有效地获取系统的内存信息：

- **引导加载程序自检**: 在引导过程中，引导加载程序可以通过特定算法扫描可用内存区域，并创建内存地图。
- **传递内存信息**: 该内存信息可以在引导完成后传递给操作系统，以便操作系统能够有效管理和利用内存。
- **标准接口**: 现代系统中，像UEFI这样的标准接口提供了获取系统内存信息的标准方式，操作系统可以通过这些接口直接获取内存布局信息。