Lab2

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

实验内容

实验一过后大家做出来了一个可以启动的系统，实验二主要涉及操作系统的物理内存管理。操作系统为了使用内存，还需高效地管理内存资源。本次实验我们会了解如何发现系统中的物理内存，然后学习如何建立对物理内存的初步管理，即了解连续物理内存管理，最后掌握页表相关的操作，即如何建立页表来实现虚拟内存到物理内存之间的映射，帮助我们对段页式内存管理机制有一个比较全面的了解。本次的实验主要是在实验一的基础上完成物理内存管理，并建立一个最简单的页表映射。

练习

练习1——理解first-fit 连续物理内存分配算法

我们接下来依次分析各个函数以及物理内存分配过程。

1.1 default_init

该函数主要用于初始化内存管理器的状态，它的主要作用是：

- **初始化空闲页链表**：调用 `list_init(&free_list)` ，对存储空闲页的双向链表 `free_list` 进行初始化。
- **清零空闲页数计数器**：将记录当前空闲内存页总数的 `nr_free` 变量设为0。

1.2 default_init_memmap

default_init_memmap主要用于初始化一个空闲内存块，包括设定其各页的属性，并将其添加到空闲链表中。参数包括 `base` （内存页起始地址）和 `n` （内存页数）。

- **内存页的初始化**：遍历 `base` 到 `base + n` 的每个 `Page` 结构，设置 `flags` 和 `property` 为0，并将引用计数 `ref` 设为0，表示这些页面是空闲的。
- **第一个页面属性**：对于该空闲块的第一个页面， `property` 设为 `n` ，表明这是一个包含 `n` 个页面的空闲块，并将 `PG_property` 标志位置1。

- **链表插入**：如果 `free_list` 为空，则将该块直接添加至链表中；否则按地址排序插入，以确保链表按地址从低到高排列。
- 更新 `nr_free` 的值，增加新加入的 `n` 个页面。

1.3 default_alloc_pages

该函数用于分配指定数量的页面（ `n` 页），采用First Fit算法从空闲链表中找到第一个满足大小要求的块。

- **遍历空闲链表**：逐个检查空闲块的 `property` ，查找大小大于等于 `n` 的第一个空闲块。
- **页面分配**：找到满足条件的空闲块后，将此块从链表中移除，并根据 `n` 的大小调整其属性。如果空闲块大小大于 `n` ，则剩余部分仍为一个空闲块，需重新计算其大小和链表位置。
- **标记页面属性**：将分配的页面从空闲状态改为非空闲，将 `PG_property` 清除，并减少 `nr_free` 中的空闲块总数。
- **返回值**：返回分配的页面地址，如果没有找到合适的块，则返回 `NULL` 。

1.4 default_free_pages

该函数用于释放指定的页面，并将其合并回空闲链表中。参数包括要释放的 `base` 页面和页数 `n` 。

- **页面属性重置**：将 `base` 到 `base + n` 范围内的每个页面重置为初始状态，清除 `PG_reserved` 标志，将 `property` 设为0，并将引用计数 `ref` 设为0。
- **添加至空闲链表**：找到链表中适当位置（按地址从低到高排列）插入新的空闲块。
- **空闲块合并**：检查新插入的空闲块两侧的块，如果它们相邻，则将其合并为一个更大的块，以提高分配效率和减少碎片。
- 更新 `nr_free` 值，增加释放的页面数量。

1.5 物理内存分配流程

在分析完上述函数后，我们对物理内存的分配流程有了较为清晰的理解。

1. **初始化**：调用 `default_init` 和 `default_init_memmap` 初始化空闲内存链表和页面属性。
2. **内存分配**：调用 `default_alloc_pages` 在空闲链表中查找第一个足够大的块，分配指定数量的页面。
3. **内存释放**：调用 `default_free_pages` 将页面返回到空闲链表，并尝试进行块合并以减少碎片。

1.6 first-fit算法改进空间

首先，面对产生碎片化问题，我们可以改进为best-fit等算法。在分配内存块时，我们可以将不同大小的空闲块放入不同的链表中，类似哈希。这样我们在分配时，根据需要的内存块大小来选择不同的链

表，能够较快速的找到对应相关的链表。同时，面对快速查找相应的块时我们也可以使用跳表或者AVL树来存储空闲块。

练习2——实现 Best-Fit 连续物理内存分配算法

Best-fit算法是遍历所有空闲块，找到一个和所需内存大小最相近的块来进行分配，这样产生的内存碎片就会很小。

在设计时，我们可以沿用first-fit相关的初始化操作，修改分配时的那一段代码即可，我们定义一个很大的变量来表示最好的块，在遍历所有空闲块时，我们进行做差，来判断内存碎片的大小，我们找到产生内存碎片最小的块来进行分配。这样我们便实现了Best-fit算法。代码实现如下：

```
static void
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: 2210751*/
        // 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: 2210751*/
            // 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
            if(base < page)
            {
                list_add_before(le, &(base->page_link));
                break;
            }
            // 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
            else if(list_next(le) == &free_list)
            {
```

```
                list_add(le, &(base->page_link));
            }
        }
    }
}
```

```
static struct Page *
best_fit_alloc_pages(size_t n) {
    assert(n > 0);
    if (n > nr_free) {
        return NULL;
    }
    struct Page *page = NULL;
    list_entry_t *le = &free_list;
    size_t min_size = nr_free + 1;
     /*LAB2 EXERCISE 2: YOUR CODE*/
    // 下面的代码是first-fit的部分代码，请修改下面的代码改为best-fit
    // 遍历空闲链表，查找满足需求的空闲页框
    // 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
    int best_size_diff = 65535; //用于存储当前找到的最佳匹配页块大小与所需大小之间的差值
    int size_diff = 0;
    while ((le = list_next(le)) != &free_list) {
        struct Page *p = le2page(le, page_link);
        //size_diff = abs(p->property - n);
        if (p->property > n)
        {
            size_diff = p->property - n;
        }
        else
        {
            size_diff = n - p->property;
        }
        if (size_diff < best_size_diff)
        {
            page = p; // 更新最佳匹配的页块指针
            best_size_diff = size_diff; // 更新最佳匹配的大小差值
            if (size_diff == 0)
            {
                break;
            }
        }
        /*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;
}
```

```
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: 2210751*/
    // 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;
    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);
        /*LAB2 EXERCISE 2: 2210751*/
        // 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));
        }
    }
}
```

2.2 优化空间

在进一步研究了自己所写的代码，我们可以增加一个内存碎片检测机制，实现一个周期性碎片检测与合并的机制。可以在 `nr_free` 页面减少到某一阈值时触发碎片整理，通过在链表中一次性合并相邻的小块，形成连续大块，以提高后续分配效率。

同时，同first-fit一样，我们可以使用多个链表存储不同大小的块，这样我们不必遍历整个链表。

扩展练习Challenge1：buddy system（伙伴系统）分配算法（需要编程）

本部分参考了往届学长的buddy system，最终较为成功的实现了该部分内容。

初始化函数部分

这一部分的核心是确保初始化的内存页都是2的整数次幂,我们采用了按大小排序的链表优化思路。

```
static void
buddy_fit_init_memmap(struct Page *base, size_t n)
{
    assert(n > 0);
    struct Page *p = base;
    for (; p != base + n; p++)
    {
        assert(PageReserved(p));
        // 清空当前页框的标志和属性信息
        p->flags = p->property = 0;
        // 将页框的引用计数设置为0
        set_page_ref(p, 0);
    }
    nr_free += n;
    // 设置base指向尚未处理内存的尾地址，从后向前初始化
    base += n;
    while (n != 0)
    {
        // 获取本轮处理内存页数
        size_t curr_n = 1;
        for(int i=0;i<fixsize(n)-1;i++){
            curr_n*=2;
        }
        // 将base向前移动
        base -= curr_n;
        // 设置此时的property参数
        base->property = curr_n;
        // 标记可用
        SetPageProperty(base);
        // 我们采用按照块大小排序方式插入空闲块链表，当大小相同时的排序策略是地址

         if (list_empty(&free_list))
         {
            list_add(&free_list, &(base->page_link));
         } else {

            list_entry_t *le;
            for(le = list_next(&free_list); le != &free_list; le =
list_next(le))
            {
               struct Page *page = le2page(le, page_link);
```

```
                // 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
                if(base < page)
                {
                    list_add_before(le, &(base->page_link));
                    break;
                }
                // 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
                else if(list_next(le) == &free_list)
                {
                    list_add(le, &(base->page_link));
                }
            }
        }
        n -= curr_n;
    }
}

```

分配函数部分

我们要向上取整（大于请求内存块的2的整数次幂），并且考虑对内存块进行分割。我们内部使用的是best-fit的逻辑，从头遍历空闲链表，找到合适的空闲块，如果此时空闲块大于请求大小，则进行分割操作，直至分配到适合的大小。

```
static struct Page *
buddy_fit_alloc_pages(size_t n) {
    assert(n > 0);

    list_entry_t *le1;
    struct Page *page1 = NULL;
    for(le1 = list_next(&free_list); le1 != &free_list; le1 = list_next(le1))
    {
        page1 = le2page(le1, page_link);
    }
    //buddy系统最大能分配的空间由最大的节点决定，而非整体空闲块数
    if(page1->property < n )
    {
            return NULL;
    }

    size_t op = fixsize(n);//得到n的最高二次幂
    size_t size=1;//得到应该分配给n的空间
    for(int i=0;i<op;i++){
```

```
        size *=2;
    }
    list_entry_t *le = &free_list;
    struct Page *page = NULL;
    // 遍历空闲链表，查找满足需求的空闲页框
    // 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){
            page = p;
            break;
        }
    }
    // 如果需要切割，分配切割后的前一块
    while (page->property > size)
    {
            page->property /= 2;
            // 切割出的右边那一半内存块不用于内存分配
            struct Page *p = page + page->property;
            p->property = page->property;
            SetPageProperty(p);
            list_add_after(&(page->page_link), &(p->page_link));
    }
    nr_free -= size;
    ClearPageProperty(page);
    assert(page->property == size);
    list_del(&(page->page_link));
    return page;
}

```

释放函数部分

合并的操作就在这里，我们将释放的内存块插入空闲链表时的规则与初始化相同。在插入之后我们将进行合并操作，我们将内存块先与前面的空闲块尝试合并，而若合并成功，合并后的块大小一定大于前面一块的大小，因此不需要再向前合并，而即使无法合并成功，我们都继续将内存块与后面的内存块合并，循环往复，直到合并到无法继续合并。

```
static void
buddy_fit_free_pages(struct Page *base, size_t n)
{
    assert(n > 0);
    size_t op = fixsize(n);//得到n的最高二次幂
    size_t size=1;
```

```
    for(int i=0;i<op;i++){
    size *=2;
    }

    struct Page *p = base;
    for (; p != base + size; p++)
    {
        assert(!PageReserved(p) && !PageProperty(p));
        p->flags = 0;
        set_page_ref(p, 0);
    }
    // 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 = size;
    SetPageProperty(base);
    nr_free += size;

    list_entry_t *le = list_next(&free_list); // 初始化 le

// 先插入至链表中
for (; le != &free_list; le = list_next(le))
{
    p = le2page(le, page_link);
    // 这里的条件修改：与初始化策略相似
    if ((base->property < p->property) || (base->property == p->property && base < p))
        break;
}
list_add_before(le, &(base->page_link));

// 合并逻辑...
// 1、判断前面的空闲页块是否与当前页块是连续的，相同大小的，如果是连续的且是相同大小的，则将当前页块合并到前面的空闲页块中
if ((p->property == base->property) && (p + p->property == base))
{
    // 2、首先更新前一个空闲页块的大小，加上当前页块的大小
    p->property += base->property;
    // 3、清除当前页块的属性标记，表示不再是空闲页块
    ClearPageProperty(base);
    // 4、从链表中删除当前页块
    list_del(&(base->page_link));
    // 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
    base = p;
    le = &(base->page_link);
}

// 循环向右合并
```

```
while (le != &free_list)
{
    p = le2page(le, page_link);
    if ((p->property == base->property) && (base + base->property == p))
    {
        base->property += p->property;
        ClearPageProperty(p);
        list_del(&(p->page_link));
        le = &(base->page_link);
    }
    // 无法合并时，退出
    else if (base->property < p->property)
    {
        // 修改 base 在链表中的位置使大小相同的聚在一起
        list_entry_t *targetLe = list_next(&base->page_link);
        while (le2page(targetLe, page_link)->property < base->property)
            targetLe = list_next(targetLe);
        if (targetLe != list_next(&base->page_link))
        {
            list_del(&(base->page_link));
            list_add_before(targetLe, &(base->page_link));
        }
        // 最后退出
        break;
    }
    le = list_next(le);
}

}
```

检测函数部分

我们编写了如下样例，并使用assert抛出错误，执行结果在后面展示：

```
static void
buddy_fit_check(void)
{
    int count = 0, total = 0;
    list_entry_t *le = &free_list;
    while ((le = list_next(le)) != &free_list)
    {
        struct Page *p = le2page(le, page_link);
        assert(PageProperty(p));
        count++, total += p->property;
    }
    assert(total == nr_free_pages());
```

```
13
14      basic_check();
15
16      struct Page *p0 = alloc_pages(26), *p1;
17      assert(p0 != NULL);
18      assert(!PageProperty(p0));
19
20      list_entry_t free_list_store = free_list;
21      list_init(&free_list);
22      assert(list_empty(&free_list));
23      assert(alloc_page() == NULL);
24
25      unsigned int nr_free_store = nr_free;
26      nr_free = 0;
27      //.....................................................
28      // 先释放
29      free_pages(p0, 26); // 32+  (-:已分配 +: 已释放)
30      // 首先检查是否对齐2
31      p0 = alloc_pages(6);  // 8- 8+ 16+
32      p1 = alloc_pages(10); // 8- 8+ 16-
33      assert((p0 + 8)->property == 8);
34      free_pages(p1, 10); // 8- 8+ 16+
35      assert((p0 + 8)->property == 8);
36      assert(p1->property == 16);
37      p1 = alloc_pages(16); // 8- 8+ 16-
38      // 之后检查合并
39      free_pages(p0, 6); // 16+ 16-
40      assert(p0->property == 16);
41      free_pages(p1, 16); // 32+
42      assert(p0->property == 32);
43
44      p0 = alloc_pages(8); // 8- 8+ 16+
45      p1 = alloc_pages(9); // 8- 8+ 16-
46      free_pages(p1, 9);   // 8- 8+ 16+
47      assert(p1->property == 16);
48      assert((p0 + 8)->property == 8);
49      free_pages(p0, 8); // 32+
50      assert(p0->property == 32);
51      // 检测链表顺序是否按照块的大小排序的
52      p0 = alloc_pages(5);
53      p1 = alloc_pages(16);
54      free_pages(p1, 16);
55      assert(list_next(&(free_list)) == &((p1 - 8)->page_link));
56      free_pages(p0, 5);
57      assert(list_next(&(free_list)) == &(p0->page_link));
58
59      p0 = alloc_pages(5);
```

```
    p1 = alloc_pages(16);
    free_pages(p0, 5);
    assert(list_next(&(free_list)) == &(p0->page_link));
    free_pages(p1, 16);
    assert(list_next(&(free_list)) == &(p0->page_link));

    // 还原
    p0 = alloc_pages(26);
    //.....................................................
    assert(nr_free == 0);
    nr_free = nr_free_store;

    free_list = free_list_store;
    free_pages(p0, 26);

    le = &free_list;
    while ((le = list_next(le)) != &free_list)
    {
        assert(le->next->prev == le && le->prev->next == le);
        struct Page *p = le2page(le, page_link);
        count--, total -= p->property;
    }
    assert(count == 0);
    assert(total == 0);
}
```

扩展练习Challenge3：硬件的可用物理内存范围的获取方法

如果OS无法具体知道硬件的可用物理内存范围，我们需要进行探测。

搜索相关资料得知，基本方法是通过BIOS中断调用来帮助完成的。通过BIOS中断获取内存可调用参数为e820h的 `INT 15h BIOS` 中断，获取内存映射地址符。其具体表示为：

```
Offset  Size    Description
00h    8字节   base address                #系统内存块基地址
08h    8字节   length in bytes             #系统内存大小
10h    4字节   type of address range      #内存类型
```

我们可以下列代码，获取内存分布信息

```
probe_memory:
//对0x8000处的32位单元清零,即给位于0x8000处的
//struct e820map的成员变量nr_map清零
```

```
                    movl $0, 0x8000
                    xorl %ebx, %ebx
//表示设置调用INT 15h BIOS中断后，BIOS返回的映射地址描述符的起始地址
                    movw $0x8004, %di
start_probe:
                    movl $0xE820, %eax // INT 15的中断调用参数
//设置地址范围描述符的大小为20字节，其大小等于struct e820map的成员变量map的大小
                    movl $20, %ecx
//设置edx为534D4150h (即4个ASCII字符“SMAP”)，这是一个约定
                    movl $SMAP, %edx
//调用int 0x15中断，要求BIOS返回一个用地址范围描述符表示的内存段信息
                    int $0x15
//如果eflags的CF位为0，则表示还有内存段需要探测
                    jnc cont
//探测有问题，结束探测
                    movw $12345, 0x8000
                    jmp finish_probe
cont:
//设置下一个BIOS返回的映射地址描述符的起始地址
                    addw $20, %di
//递增struct e820map的成员变量nr_map
                    incl 0x8000
//如果INT0x15返回的ebx为零，表示探测结束，否则继续探测
                    cmpl $0, %ebx
                    jnz start_probe
finish_probe:
```

具体参考链接为内存探测。

此外，还了解到，OS在初始化物理内存管理时，也可以通过 RAM 的 SPD（Serial Presence Detect）读取物理内存大小。

最后，经过相关的搜索，还了解到以下方法也是可以使用的：

1. **通过 ACPI 表获取内存信息**：
    - 高级配置与电源接口 (ACPI) 提供了一系列描述系统硬件资源的表格。操作系统可以解析 ACPI 的系统描述表，例如 SRAT (System Resource Affinity Table) 和 SLIT (System Locality Information Table)，以识别可用的内存区域和系统硬件拓扑。
    - ACPI 的内存映射信息在现代系统中较为常用，特别是在服务器和多处理器系统上，能够提供更详细的内存分布信息。
2. **从内存控制器读取内存信息**：
    - 某些硬件架构允许操作系统直接查询内存控制器，以获取物理内存的大小和分布。例如，x86 架构的一些芯片组包含寄存器，可以反映内存条的容量和布局。操作系统可以通过读取这些寄存器来获得内存的总量。

- 然而，这种方法通常与硬件紧密耦合，仅适用于支持直接访问内存控制器信息的系统，且不同的架构可能需要不同的访问方法。

总结

在该实验中，我们学习并理解了页表的建立和使用，物理内存的管理方法，页面分配算法。OS中相关的重要知识点，我们也在此次实验中得到了较为充分的理解和深入。对伙伴系统，几个页分配算法。在OS课程中，我们只是理解这些算法的实现流程。在实验编写时，我们对内核和物理内存的管理如何成功才有了更深的理解。