# Lab2 物理内存和页表

2313202 葛佳琦

2312321 吴昊然

2312554 袁宇菡

## 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

在实验一做出的一个可以启动的系统的基础上，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。本次实验将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较为全面的了解。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 1：理解 first-fit 连续物理内存分配算法（思考题）

first-fit 连续物理内存分配算法作为物理内存分配一个很基础的方法，需要同学们理解它的实现过程。请大家仔细阅读实验手册的教程并结合 kern/mm/default_pmm.c 中的相关代码，认真分析 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：

你的 first fit 算法是否有进一步的改进空间？

First-Fit 算法的思想：

- 维护一个空闲物理内存块的链表。
- 每次分配内存时，从链表头开始扫描，找到第一个能满足请求大小的空闲块。
- 如果这个空闲块比所需更大，就把它切分成两部分：

  ---前半部分分配出去；

  ---后半部分继续挂在空闲链表中。
- 如果刚好大小一致，就直接分配整块，并从链表删除该块。
- 释放时，将被释放的物理页块重新插入链表，并尝试与前后块合并，避免碎片化。

- **default_init**

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

**功能：**

1.初始化空闲页链表 free_list,通过 list_init(&free_list) 把链表头初始化为空。此时链表为空，没有任何 free block。free_list 是全局 free_area_t 里的成员，是一个双向循环链表。

2.初始化全局空闲页计数器 nr_free，把空闲页数量设为 0，后续 default_init_memmap 会增加它。

即主要功能就是物理内存管理器的“开机初始化”，实现***链表初始化 + 空闲页计数器归零***，这是整个内存分配器的初始化入口；

- **default_init_memmap**
  - 将 *[base, base+n)* 范围的所有页标记为空闲：
    - *flags* 清零
    - *ref* 设为 0（表示无人使用）
  - 第一页的 *property* 设为 *n*，其余页 *property = 0*

- 插入 *free_list*，**保持链表按地址升序排序**
- *nr_free += n*

  nr_free 表示空闲页页数

**作用**：初始化 base 开始的 n 个物理页，加入空闲链表，建立起“初始空闲内存块”的链表节点。

```
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);
    }//遍历每页，把 flags/property/引用计数清零
    base->property = n;
    SetPageProperty(base);
    nr_free += n;//全局空闲页计数更新

    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));
            }//遍历 free_list 按地址查找插入位置,保持 free_list 按物理地址升序
        }
    }
}
```

**功能：**

1.把一段连续物理页标记为“可用”

2.记录空闲块的大小

3.把它插入 free_list 链表，以便 First-Fit 分配算法使用

简而言之就是**告诉操作系统：这里有一块空闲内存**。

**关键概念**

- **Page 结构**

在 ucore 中，每个物理页对应一个 struct Page，里面重要字段：

flags:标记页属性，例如 PageReserved，PageProperty

property:仅块头存储连续空闲页数量

ref：被引用次数

page_link:链表节点，连接 free_list

- **链表和块头**

块头（head page）：property = 块大小，PG_property=1

非块头页：property=0，只是链表节点，不能单独分配

**这就是 First-Fit 分配逻辑的基础：只看块头的 property 来判断能否分配**

**property 只在块头有效，非块头页不记录大小，只参与链表操作**

- **default_alloc_pages**

从 free_list 头开始扫描，找到第一个 p->property >= n 的空闲块（即满足请求的空闲块）从链表中删除该块，如果 p->property > n（块大于请求），将剩余部分拆成新块重新插入链表，更新 nr_free -= n 返回起始 page 地址

**作用**：根据需要分配的页面数量，从空闲链表中取出第一个可用的连续块，并分配它。

- **default_free_pages**
  - 将 *[base, base+n)* 重新标记为空闲页
  - 插入 *free_list*，按地址顺序插入
  - 检查前后块：
    - 如果前块的结束地址紧挨 *base*，则合并
    - 如果后块紧挨 *base+n*，也合并
  - 更新 *nr_free += n*

**作用**：清理释放页的标记，按地址顺序插入 `free_list`，尝试合并与前后相邻空闲块，减少外部碎片。

**First Fit 改进空间：**

1. 这种分配方法会让空间碎片化，可能会出现在空间总量足够的情况下却没有足够大的页面可用的情况；
2. 为了能够减少碎片化，在释放页的时候会检查块前后地址是否相邻来进行合并，所以空闲块链表是按物理地址升序排序的，但是这同时也使得在找寻大小相近的块时需要线性扫描链表，搜索的效率较低。
3. 现在这种分配方式没有考虑页号对齐。
4. 搜索优化：当前实现每次分配都遍历链表，可以考虑按块大小分级减少搜索。
5. 碎片优化：当前依赖释放时合并，对小块频繁分配仍可能产生碎片。

## 练习 2：实现 Best-Fit 连续物理内存分配算法（需要编程）

在完成练习一后，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，算法的时空复杂度不做要求，能通过测试即可。 请在实验报告中简要说明你的设计实现过程，阐述代码是如何对物理内存进行分配和释放，并回答如下问题：

你的 Best-Fit 算法是否有进一步的改进空间？

Best-Fit 算法是在空闲块中找到 **最小但足够大的块** 来满足分配请求，从而尽量减少外部碎片。

在具体代码实现中，Best-Fit 的实现过程与 First-Fit 的主要不同在于 **best_fit_alloc_pages** 的函数实现

主要的实现流程在注释中已经标识出来了，核心的部分在找到最小空闲块的部分，遍历整个空闲页链表，

对每个空闲块 *p*：

- 如果 *p->property >= n*，说明它能满足请求。
- 如果 *p->property < min_size*，说明它比之前找到的块更小，更接近请

求大小。

- 更新 *best_page = p*，并更新 *min_size = p->property*。

  经过以上循环过程就可以找到最合适分配的空闲块。

之后保存 *best_page* 的前一个链表节点，将其从链表中删除出去。如果找到的空闲块比所需要的要多，就拆分这一块，并将拆分后得到的块链入空闲链表。

逻辑：

1. 遍历整个空闲链表。
2. 找到**最小的满足 p->property >= n 的块。**
3. 将其分配，并在链表中删除。
4. 若块大于请求，将剩余部分拆分并重新加入链表。

特点：

相比 First-Fit，减少了内存浪费，因为选的是最小的可用块。但是仍然需要遍历整个链表，搜索时间较长。

对物理内存进行释放的代码实现与 First-Fit 类似，

1. 插入空闲链表
2. 尝试合并前后块
3. 更新 nr_free

**运行截图如下，通过 make grade 测试**

```
uki@uki-virtual-machine:~/riscv/labcode/lab2$ make grade
>>>>>>>>>> here_make>>>>>>>>>>>
gmake[1]: Entering directory '/home/uki/riscv/labcode/lab2' + cc kern/init/entry
.S + cc kern/init/init.c + cc kern/libs/stdio.c + cc kern/debug/panic.c + cc ker
n/driver/console.c + cc kern/driver/dtb.c + cc kern/mm/best_fit_pmm.c + cc kern/
mm/default_pmm.c + cc kern/mm/pmm.c + cc libs/printfmt.c + cc libs/readline.c +
cc libs/sbi.c + cc libs/string.c + ld bin/kernel riscv64-unknown-elf-objcopy bin
/kernel --strip-all -O binary bin/ucore.img gmake[1]: Leaving directory '/home/u
ki/riscv/labcode/lab2'
>>>>>>>>>> here_make>>>>>>>>>>>
<<<<<<<<<<<<<<<< here_run_qemu <<<<<<<<<<<<<<<<<<
try to run qemu
qemu pid=14157
<<<<<<<<<<<<<<<< here_run_check <<<<<<<<<<<<<<<<<<
  -check physical_memory_map_information:    OK
  -check_best_fit:                           OK
Total Score: 25/25
```

**改进方向**：可以将原本的空闲链表改进为多级空闲链表，按块的大小来分级，

每个链表每个链表存放大小相近的块，内部按物理地址排序。

代码如下

```
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: 2313202 2312321 2312554*/
    // 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
    struct Page *best_page = NULL;
    while ((le = list_next(le)) != &free_list) {
        struct Page *p = le2page(le, page_link);
        if (p->property >= n && p->property < min_size) {
            best_page = p;
            min_size = p->property;
        }
    }
    if (!best_page) return NULL; //遍历free_list找到最小空闲块
    list_entry_t* prev = list_prev(&(best_page->page_link));
    list_del(&(best_page->page_link));

    if (best_page->property > n) {
        struct Page *p = best_page + n;
        p->property = best_page->property - n;
        SetPageProperty(p);
        list_add(prev, &(p->page_link));
    }
    ClearPageProperty(best_page);
    nr_free -= n; //分配
    return best_page;
}
```

## 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。

详细内容见设计文档 **lab2_c1.pdf**

## 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。

在完成此 challenge 时能生成内核镜像，但观察不到应有的 satp 输出，故失败。

## 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？

- 自我探测方法

1. 写-读测试探测：操作系统按页遍历物理内存，对每个页写入特定数据并读回验证。可用内存页能够正确存储数据，而不可用或不存在的页会导致异常或数据错误。这种方法完全由 OS 自行完成，无需依赖外部信息，但效率低，且操作不慎可能覆盖已有重要数据或访问特殊硬件区域。

2. 硬件寄存器/控制器查询：通过访问主板或内存控制器提供的寄存器（如 DDR

控制器配置寄存器），操作系统可直接获取可用内存的起止地址和大小。该方法比写-读测试更安全、高效，但依赖硬件支持，且不同平台实现差异较大。

3. 异常访问探测：操作系统尝试访问特定内存地址，如果触发异常（如 CPU 异常或总线错误），则该地址不可用；访问成功则可用。该方法无需写入数据，安全性较高，但实现复杂，需要操作系统能够捕获异常并正确恢复。

- 依赖外部信息的方法（现代 OS 常用）

1. Bootloader 或固件提供信息：系统启动阶段，Bootloader 或固件（如 BIOS）扫描物理内存，并将可用内存信息传递给操作系统。OS 启动时直接读取这些信息，无需自己探测，效率高、安全性好，但依赖外部提供信息。

2. ACPI 表或其他硬件描述表：操作系统可读取 ACPI（高级配置与电源接口）表或类似硬件描述表中的内存信息，获取可用内存区间。标准化且效率高，但需要硬件和固件支持。

3. 外设/DMA 试探：通过配置 DMA 控制器或外设访问物理内存，如果传输失败，则该内存地址不可用。可间接检测内存区域，避免直接写入内核数据，但实现复杂，需要硬件支持。