# 华东师范大学计算机科学与技术学院上机实践报告

| | | |
|---|---|---|
| **课程名称**：操作系统 | **年级**：2022 级 | **上机实践日期**：2024 年 4 月 16 日 |
| **指导教师**：李东 | **姓名**：岳锦鹏 | **学号**：10213903403 |
| **实验名称**：实验二　物理内存管理 | | |

## 一、 研读理解相关代码后，回答以下问题：

### 1、 uCore 如何探测物理内存布局，其返回结果 e820 映射结构是什么样的？分别表示什么？

首先探测物理内存布局可以通过 BIOS 中断或直接探测，其中 BIOS 中断调用方法通常只能在实模式下完成，直接探测的方法必须在保护模式下完成，这里 $\mu$Core 是在实模式下通过 BIOS 中断调用完成的。具体可以分为以下三步骤：

(1) 设置一个存放内存映射地址描述符的物理地址（在此为 0x8000）；

(2) 将 e820 作为参数传递给 INT 15h 中断；

(3) 通过检测 eflags 的 CF 位来判断探测是否结束。如果 CF 位为 0，则表示探测没有结束，那么就需要设置存放下一个内存映射地址描述符的物理地址，返回步骤 2 继续进行；否则物理内存检测就此结束。

其返回结果 e820 的结构为：

```c
struct e820map {
    int nr_map;
    struct {
        uint64_t addr;
        uint64_t size;
        uint32_t type;
    } __attribute__((packed)) map[E820MAX];
};
```

其中 `nr_map` 表示内存映射地址描述符的数量，`addr` 表示系统内存块基地址，`size` 表示系统内存块大小，`type` 表示内存类型。

### 2、 uCore 中的物理内存空间管理采用什么样的方案？跟我们理论课中的哪个方案相似？有何不同之处？

$\mu$Core 中的物理内存管理采用段页式系统，但是简化了分段机制，将逻辑地址直接恒等映射到线性地址，之后使用分页机制将线性地址通过分页映射到物理地址。由于使用了分页机制，因此需要对空闲物理块管理，对空闲物理块的管理使用了链表来管理，链表按照地址排序。

跟我们理论课中的“段页式存储管理”和“使用链表管理存储空间”的方案相似，不同之处在于分段机制简化了，并且只使用链表管理了空闲物理块（已分配的物理块在页表中有记录，所以不需要管理）。

3、 Page 数据结构中每个字段含义与作用是什么？如何表示某物理块分配与否？字段 property 的作用是什么？如何表示 property 是否有效？

```
struct Page {
    int ref;                    // page frame's reference counter
    uint32_t flags;             // array of flags that describe the status of the page
    ↪ frame
    unsigned int property;      // the num of free block, used in first fit pm manager
    list_entry_t page_link;     // free list link
};
```

ref 字段表示此物理块被引用的个数，如果大于 1 代表这个物理块对应的可能是共享内存；flags 的 0 位表示此物理块是否已被分配，0 位为 1 代表已被分配；1 位代表此描述符的 property 字段是否有效，1 位为 1 代表有效。property 只有在此物理块是空闲块时才有效，表示从此物理块开始连续的空闲物理块的数量。

用 flags 的 0 位表示此物理块是否已被分配，0 位为 1 代表已被分配。

property 的作用是表示从此物理块开始连续的空闲物理块的数量。

用 flags 的 1 位表示 property 是否有效，1 位为 1 代表有效。

4、 uCore 现有代码已实现的物理块分配首次适应算法以及物理块回收算法有没有问题或者错误？如有的话，请简述相关的问题或者错误，并修改相应的代码。

default_init_memmap 函数中的 `p->flags = 0;`，这里应该是把 flags 的 0 位设置成 0，而不是把整个 flags 都设置成 0。所以应该修改成 `p->flags &= ~1;`。

5、 你认为在 uCore 的四个物理块基本分配方案基础上有没有进一步优化的可能？如有的话，请简要说明你的相关优化方案。

四个物理块基本分配方案难道是 FF（first fit，首次适应）、BF（best fit，最佳首次适应）、WF（worst fit，最坏首次适应）、NF（next fit，循环首次适应）？有进一步优化的可能，比如合并空闲的内存块：在系统空闲时可以检测所有已分配的物理块，将已分配但最近未使用的物理块合并到相邻的位置（最近使用的物理块不能乱动，不然会影响性能还可能出现问题）。

6、 get_pte、get_page 函数的作用是什么？它们的输入与返回分别是什么？

```
pte_t *get_pte(pde_t *pgdir, uintptr_t la, bool create)
```

get_pte 的作用是根据逻辑地址返回页表指针，它的输入为页目录的起始地址、逻辑地址、页表是否已被调入内存，返回为页表指针。

```
struct Page *get_page(pde_t *pgdir, uintptr_t la, pte_t **ptep_store)
```

get_page 的作用是根据逻辑地址返回对应的物理块描述符，它的输入为页目录的起始地址、逻辑地址、以及可能需要存储的页表指针的地址，返回为物理块描述符。

## 二、 程序设计与实现的基本思路

1、 关于物理内存管理的部分是在 kern/mm/中（这里的 mm 应该是 memory map 内存映射的意思？），循环首次适应算法是分配物理块的算法，所以观察到分配物理块的部分在 default_pmm.c 中实现，那么只需要关注这个文件的内容。

2、 原先的代码中实现的是首次适应算法（first fit），这里要改成循环首次适应算法（next fit），只需要记录下每次分配的指针，每次从该指针的位置开始继续查找下一个空闲块。这里使用了 list_entry_t *alloc_le = &free_list; 作为全局变量来记录每次分配的指针。（为什么不用静态局部变量？是因为它的测试函数的问题，后面会提到）这里的 alloc_le 和 default_alloc_pages 中的 le 是同样的作用，只是全局变量名字不能和已有的一样所以改了个名。

3、 这里新建的物理块分配函数使用 next_fit_alloc_pages 命名和原来的区分，在这个函数中首先需要初始化 alloc_le 指向链表头结点，但这个初始化又必须执行且只执行一次，所以这里使用了分支语句，

```
if (alloc_le == &free_list)
    alloc_le = list_next(&free_list);
```

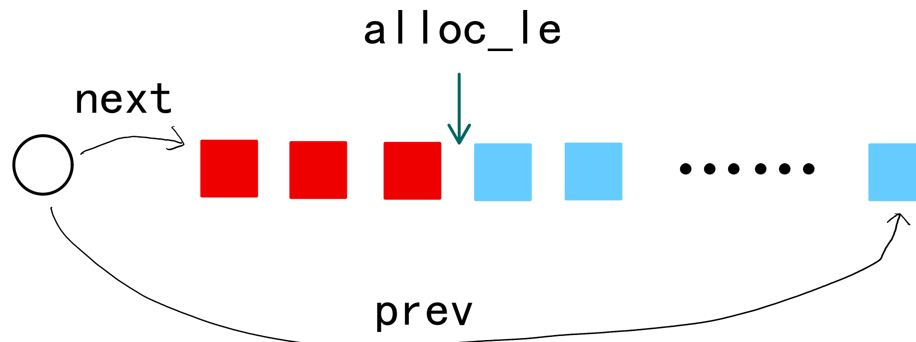
4、 接下来就开始从 default_alloc_pages 中的循环着手，原先的循环条件是 while((le=list_next(le)) != &free_list)，表示每次查看下一个链表表项，直到到达末尾（由于是循环链表，所以末尾就是头结点即&free_list）。那么这里就需要修改成循环到下一次遇到 alloc_le。假设链表共有 $n$ 项，那么最多需要执行 $n+1$ 次循环，并且最后一次循环不满足条件退出。那能不能直接固定循环 $n$ 次呢？这里是不行的，还是它的测试函数的问题，也在后面提到。因此首次循环和最后一次循环，循环变量指向的都是同一个位置，但是首次循环需要满足条件，最后一次循环需要不满足条件，这应该怎么实现呢？这就是下面要介绍的中途改变循环结束指针的方式。

```c
static struct Page *
next_fit_alloc_pages(size_t n) {
    list_entry_t *start = alloc_le;
    list_entry_t *end = &free_list;
    while(1) {
        if (alloc_le == end) { // 最多两次触发此条件
            if (end != start) { // 第二次触发 le == end 的时候这里就不满足了
            end = start; // 如果循环到链表末尾了就把结束的指针改成循环开始的位置
            alloc_le = list_next(alloc_le); // 并且这个 &free_list 不应该被分配
            continue;
            }
            break; // 这时候就是真正的循环完了也没有剩余空间
        }
        ...
        alloc_le = list_next(alloc_le);
    }
    return NULL;
}
```

首先，用 `start` 记录循环指针开始的位置，用 `end` 记录头结点的位置，之后进入循环，在首次循环到末尾的时候（即`if (alloc_le == end)`，把 `end` 修改到 `start` 的位置，之后第二次触发`if (alloc_le == end)` 的时候，就可以退出循环了。也就是把循环遍历分成了两段，假设循环指针当前位置为 $k$，共有 $n$ 个链表节点，这就是分成了 $k \cdots n$ 的前一段和 $1 \cdots k$ 的后一段。

5、 上文多次提到原先的测试代码的问题，这里它的测试代码在 `default_check` 和 `basic_check` 这两个函数中，这两个函数的目的是检测内存的分配与回收的方案是否正确，具体方法是多次分配回收，例如先分配 3 个物理块，之后把空闲物理块链表清空，这时候空闲物理块是多少？已经清空了所以是 0 个对不对？那么这时候要再分配一个物理块呢？应该是无法分配的是吧，但这里就出现问题了，循环首次适应的指针这时候在哪呢？还在之前分配了的位置，所以实际上这时候还是能分配到空闲物理块的，这就出现问题了，它的 `assert(alloc_page() == NULL);` 就不通过了。

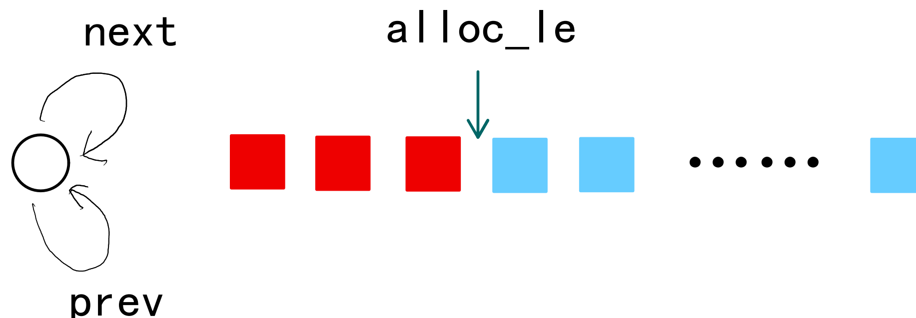
分配了 3 个物理块后的情况如下图所示，红色矩形代表已经被分配的物理块，蓝色矩形代表未被分配的物理块，左侧的圆形代表链表头结点，prev 和 next 表示头结点的前驱和后继节点。此时已经分配了 3 个物理块，所以 `alloc_le` 记录的位置在第三个物理块后。




但是之后出现了 `list_init(&free_list);` 这一行，这个函数的定义如下

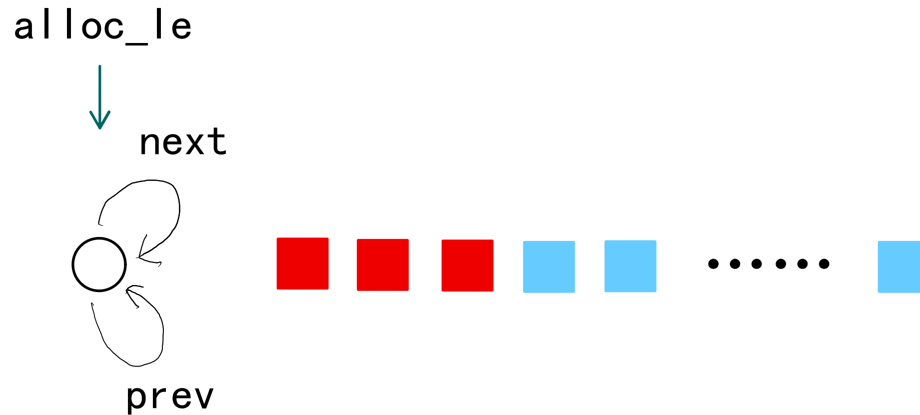
```
/* *
 * list_init - initialize a new entry
 * @elm:        new entry to be initialized
 * */
static inline void
list_init(list_entry_t *elm) {
    elm->prev = elm->next = elm;
}
```

可见此函数只是把链表的前驱节点和后继节点都指向了自己，也就变成了下图：




这时候似乎没什么问题，但注意它的清空不彻底，再分配物理空间的时候，`alloc_le`

还在原先的位置，所以仍然能分配成功，这就导致 `assert(alloc_page() == NULL);` 不通过了。所以解决方案也很简单，只需要把 `alloc_le` 也指向头结点就行了，如下图所示：




还要注意它在清空链表之后还有个恢复的操作，所以这个 `alloc_le` 也需要恢复，而且在 `default_check` 和 `basic_check` 中都有这样的操作，因此两个函数，每个函数一次保存一次恢复，所以需要添加四行 `alloc_le = &free_list;`。

6、 时间仓促，如有错误敬请指出（在下方的链接中）。

## 三、 代码

1、 https://gitea.shuishan.net.cn/10213903403/os_kernel_lab

2、 也可以看上传的附件。