Lab 2 Extra

准备工作：创建并切换到 `lab2-extra` 分支

请在**自动初始化分支后**，在开发机依次执行以下命令：

```
$ cd ~/学号
$ git fetch
$ git checkout lab2-extra
```

初始化的 `lab2-extra` 分支基于课下完成的 `lab2` 分支，并且在 `tests` 目录下添加了 `lab2_malloc` 样例测试目录。

题目背景

在 Lab2 课下实验中，我们实现了粒度为 4KB 的页式内存管理，在这里，我们将要实现简化的 **malloc** 和 **free** 函数。

malloc 函数将会在堆区域申请一个指定大小的内存空间，并且搭配 free 函数可以实现运行中的内存回收。malloc 在分配内存空间时，将该空间的几个字节作为管理内存块的元数据，元数据后紧挨着分配给用户的内存空间。在我们的程序中，元数据为一个特别设计的结构体 **MBlock**，用于记录被切割的内存空间。

MBlock 结构体介绍：

```
struct MBlock {
    MBlock_LIST_entry_t mb_link; // 链表控制块（该属性占用 8 字节）

    u_int size;     // 该元数据所管理的内存空间大小（该属性占用 4 字节）
    void *ptr;      // 该元数据管理内存空间的首地址（即 data 位置，仅作为魔数使用）（该属性占
    u_int free;     // 该元数据所管理内存空间是否空闲，1 为空闲，0 为占用（该属性占用 4 字节
    u_int padding; // 填充，用于将结构体内存大小对齐到 8 字节，（该属性占用 4 字节）
    char data[];  // 该元数据管理的内存空间，仅用于表示内存空间的首地址，不带有实际含义
};
```

其中 `data[]` 在结构体中并不占用内存空间，如果你用 `sizeof(struct MBlock)`，会发现，结构体的大小只有 24字节，正好是**前五项参数**的大小，这里的

`data[]` 是 C 语言的柔性数组，表示在结构体后的任意长度空间，可以看作表示所管理的内存空间的数组。

那么 `*ptr` 又是什么呢，为什么已经存在 `data[]` 表示内存空间地址的情况下，还需要 `*ptr` 这一项呢？实际上， `*ptr` 是作为魔数存在的，通过使 `ptr = data` 将内存空间地址保存在元数据内部，而 `free` 时可以再次检查 `ptr == data` ，确认当前地址是否是由 `malloc` 分配得到。

当进行一次分配时，程序会在 MBlock 组成的链表中找到合适的空闲 MBlock，并切割出足够的空间，用于生成记录新空间的元数据 MBlock 和对应分配出的内存空间。换句话说，MBlock 作为分割符，在一段连续的内存空间中，切割出了一系列内存空间。

通过这种方式，可以实现任意粒度的内存分配，以此来减少页中产生的内碎片，注意，元数据和分配给用户的内存空间应当交替紧邻。

题目描述

在本题中，你需要对 MOS 的内核空间 4MB 虚拟内存提供简化的 malloc 和 free 函数。

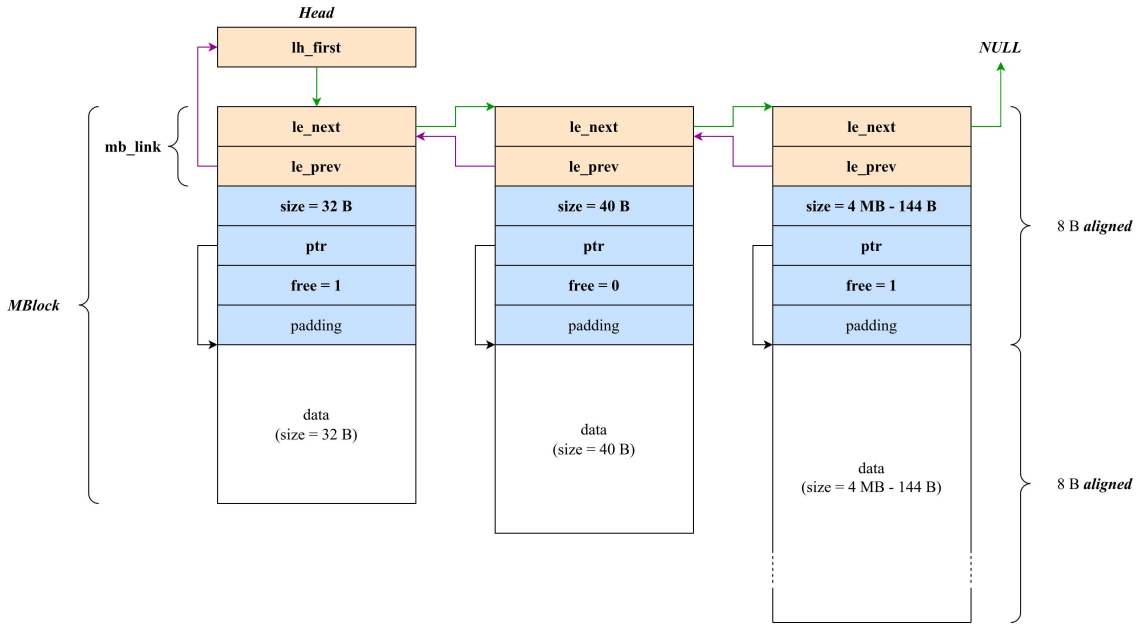
malloc 函数在收到分配内存的请求时，会将请求值向上对齐到 8 字节的倍数，并且从链表中找到合适的元数据进行分配，在这里，我们选择 **First-Fit** 方法，即选择遇到的第一个符合要求的元数据。

当遇到一个元数据时，检查其空间大小是否足以分配需要的内存空间

- 如果是，则维护元数据，将需要的空间返回给用户，并通过新建元数据的方式维护未被使用的剩余空间
- 如果不足，继续遍历链表，寻找下一个元数据
  free 函数通过 MBlock 中的 ptr 属性检查地址是否是当初被 malloc 分配出去的地址，之后标记为可用，**合并相邻空闲空间**。

以进行两次分配和一次释放后的内存为示例，其调用顺序和参数为：

```
void *p1 = malloc(32);
void *p2 = malloc(40);
printk("p1:%x p2:%x\n", (int)p1, (int)p2);
free(p1);
```



方法的实现要求与细节请参看**题目要求**部分。

题目要求

在本题中，你需要使用对内核空间的 `0x80400000 - 0x80800000` 实现动态内存分配和回收。

- 元数据结构将会在 `include/malloc.h` 给出，其中结构体有效大小为 24 字节，最后 `data` 指针不需要实际分配内存，仅表示分给用户的空间位置。
- 初始将在 `0x80400000` 处声明一个基础元数据，其可用空间大小为 4MB 减去 24B，之后从这里作为起点。

同时，我们将提供部分代码（请参看**实验提供代码**部分），你需要将其粘贴至 `kern/pmap.c` 之后，并补全或者实现如下函数：

内存的分配（ `void* malloc(size_t size)` ）

本函数的功能为：

- 使用 First-Fit 分配大小不低于 `size` 字节的内存空间：
  - 分配成功时，将分配的内存区间的**首地址**返回。
  - 分配失败时，返回值为 `NULL` 。
- 分配的内存区间需要满足以下条件：
  - 内存区间空闲，也即未被分配。
  - 区间应当紧凑，区间和元数据之间不存在无意义的空白区域。
  - 内存区间大小为不小于 `size` 的最小的 8 字节倍数。
  - 分配地址应当**以 8 对齐**。

以下是 `malloc` 函数的参考实现方案（你也可以自行实现本函数，但必须保证满足函数定义与功能约束）：

1. 计算需要分配的 8 字节对齐的字节数。
2. 从 `0x80400000` 处开始，遍历链表，直到找到剩余空间大于等于 `size` 的元数据。
3. 当需要分配的 `size` 小于等于剩余空间大小时：
    i. 若剩余空间大小，除去分配给用户的内存空间 `size` ，剩余大小不足分配一个新的的元数据与至少 8 字节的空闲空间，则剩余空间一齐分配给调用者。
    ii. 若剩余空间大小，除去分配给用户的内存空间 `size` ，仍剩余一个元数据的空间与对应至少 8 字节的空闲空间可以使用，则在分配的数据空间 `size` 后，再建立新的元数据，并维护元数据。
4. 若未找到满足条件的元数据，则返回 `NULL` 。
5. 将为用户分配空间的首地址返回用户（data 指针位置），而不是 MBlock 的地址。

注意：

- 本函数返回的内存区间必须从 `0x80400000 - 0x80800000` 中取得，并且相邻空间不互相覆盖。
- 保证调用函数时参数 `size` 不为 0，也不超过 1MB。
- 不需要考虑清空对应物理页面中的数据。
- 不应存在不被元数据管理的内存空间，也不应存在管理空间大小为 0 的元数据。
- 结合 `free` 函数，可能会先后申请总和超过 4MB 的内容。

内存的释放（ `void free(void* p)` ）

本函数的功能为：

- 释放 `p` 对应的内存区间， `p` 应当是 `malloc` 分配的区间首地址，如果不是首地址，则**无需释放**。
- 如果释放后，相邻有其他空闲空间，则**合并相邻内存空间**。

以下是 `free` 函数的参考实现方案（你也可以自行实现本函数，但必须保证满足函数定义与功能约束）：

1. 判断当前地址 `p` 是否在合理的范围内 `[HEAP_BEGIN + MBLOCK_SIZE, HEAP_BEGIN + HEAP_SIZE]`
2. 通过当前地址 `p` 减去 `MBLOCK_SIZE` 得到 `MBlock` 应在位置。
3. 判断当前 `MBlock` 是否合法，可以通过元数据中指向首地址的指针 `ptr == data` 判断（保证测试中可以使用）。
4. 当需要释放空闲区间，且相邻存在空闲区间时：
    i. 若后一个区间空闲，将后元数据清除，空间大小加到当前区间元数据中，修改指针位置，设为空闲。

ii. 若前一个区间空闲，将当前元数据清除，空间大小加到前区间元数据中，修改指针位置。

5. 当前后区间均占用：
   i. 将当前空间元数据设为空闲。

注意：

- 当需要释放的内存区间及其相邻空间均空闲时，必须立即将其合并为更大的空闲区间。
- 参数 `p` 可能存在并非通过 `malloc` 分配得到的地址，此时 `p` 不应被释放。
- `malloc.h` 中定义的 `MBLOCK_PREV` 是为本道题简化的前向链表方法，通过 `MBLOCK_PREV(elm, field)` 可以得到 MBlock 链表前一个元素的指针。
- 可以用 `LIST_FIRST` 或者 `MBLOCK_PREV` 是否指向链表头判断是否是头元素。

任务总结

在提交前，你需要完成以下任务：

- 完成 `malloc` 函数。
- 完成 `free` 函数。
- **修改** `kernel.lds` **将** `end` **设置为** `0x80800000`

实验提供代码

请将本部分提供代码附加在你的 `kern/pmap.c` 的尾部，然后开始完成题目。

```
#include <malloc.h>

struct MBlock_list mblock_list;

void malloc_init() {

    printk("malloc_init begin\n");

    LIST_INIT(&mblock_list);

    struct MBlock *heap_begin = (struct MBlock*) HEAP_BEGIN;

    printk("heap_begin: 0x%X\n", heap_begin);

    heap_begin->size = HEAP_SIZE - MBLOCK_SIZE;
    heap_begin->ptr = (void*) heap_begin->data;
    heap_begin->free = 1;

    LIST_INSERT_HEAD(&mblock_list, heap_begin, mb_link);
```

```
    printk("malloc_init end\n");

}

void *malloc(size_t size) {
    /* Your Code Here (1/2) */
}

void free(void *p) {
    /* Your Code Here (2/2) */
}
```

样例说明

对于下列样例：

```
void *p1 = malloc(32);
void *p2 = malloc(40);
printk("p1:%x p2:%x\n", (int)p1, (int)p2);
free(p1);
```

其应当输出：

```
p1:80400018 p2:80400050
```

即上面所提供的内存示例图

本地测试说明

你可以使用：

- `make test lab=2_malloc && make run` 在本地测试上述样例（调试模式）
- `MOS_PROFILE=release make test lab=2_malloc && make run` 在本地测试上述样例（开启优化）

或者在 `init/init.c` 的 `mips_init` 函数中自行编写测试代码并使用 `make && make run` 测试。

如果样例测试中输出了如下结果，说明你通过了本地测试。

```
malloc_test() is done
```

提交评测